Micro-control de unidades en combates de StarCraft

Trabajo realizado por:

Miguel Ascanio Gómez *Alberto Casas Ortiz* *Raúl Martín Guadaño*

Trabajo de fin de grado del grado en Ingeniería informática

Facultad de Informática

Universidad complutense de Madrid

Curso 2015/2016

Director:

Antonio Sánchez Ruiz-Granados

Departamento de Ingeniería del software e Inteligencia Artificial

El código de nuestro framework se encuentra en el siguiente repositorio de GitHub:

https://github.com/TFG-StarCraft/TFG-StarCraft

El código se encuentra bajo la licencia libre GPLv2.0:

GNU General Public License v2.0

Palabras clave: inteligencia artificial, aprendizaje automático, StarCraft, aprendizaje por refuerzo, q-learning, framework, videojuegos, BWAPI, trabajo de fin de grado, conductismo

Key words: artificial intelligence, Machine learning, StarCraft, reinforcement learning, q-learning, framework, videojuegos, BWAPI, final degree project, behaviorism

AUTORIZACIÓN PARA LA DIFUSIÓN DEL TRABAJO FIN DE GRADO Y SU DEPÓSITO EN EL REPOSITORIO INSTITUCIONAL E-PRINTS COMPLUTENSE

Los abajo firmantes, alumno/s y tutor/es del Trabajo Fin de Grado (TFG) en el Grado en ……de la Facultad de ……., autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el Trabajo Fin de Grado (TF) cuyos datos se detallan a continuación. Así mismo autorizan a la Universidad Complutense de Madrid a que sea depositado en acceso abierto en el repositorio institucional con el objeto de incrementar la difusión, uso e impacto del TFG en Internet y garantizar su preservación y acceso a largo plazo.

Periodo de embargo (opcional):

- ☐ 6 meses
- ☐ 12meses

TÍTULO del TFG: ……
……
Curso académico: 20….. / 20…..

Nombre del Alumno/s:
……
……

Tutor/es del TFG y departamento al que pertenece:
……
……
……

Firma del alumno/s

Firma del tutor/es

Agradecimientos:

A todas aquellas personas que nos han apoyado y nos han dado ánimos para llegar hasta aquí, incluyendo amigos, familiares y profesores.

A nuestro profesor y tutor del trabajo Antonio Alejandro Sánchez Ruiz-Granados por aceptar y hacer más interesante nuestra investigación.

A los creadores de BWAPI y BWMirror por el trabajo realizado que han hecho más ameno el trabajo en el entorno de Starcraft.

Contenido

Abstract

One of the main goals in artificial intelligence is to achieve entities that act autonomously against different situations with no need of a human being deciding what to do. In the case in which we are in an immovable and static world, it is possible to code an algorithm that solve problems efficiently, but when we are in a variable world, there may arise new problems to which our algorithm cannot work, in that case, we need our entity to learn to confront those problems.

An example of variable world can be the world of StarCraft [14], a strategy game in which we dispose of a base and a serie of units that we must control with the goal of destroying the enemy base. In that kind of environment, it is occasionally difficult to manage the different situations to reach our goal, that's why the main objective of this project is to code a bot which learn to control groups of units and to confront the different situations, allowing the bot to discover strategies to win the game learning from its own experience.

According to the fact that we don't know in which situations we'll be, it seems to be ideal to use the reinforcement learning, which allows the bot to learn interacting with the environment from where initially it has no data through experience, using for that task a system of compensation of the different actions carried out in function of its consequences.

Resumen

Uno de los principales objetivos de la inteligencia artificial es conseguir entidades que actúen de forma autónoma ante diferentes situaciones sin necesidad de que haya un humano detrás decidiendo la siguiente acción. Cuando nos encontramos en un mundo estático e inmóvil, es posible conseguir crear un algoritmo que resuelva los problemas de manera eficaz, pero cuando el mundo en el que nos encontramos varía, pueden surgir nuevos problemas nuevos para los que el algoritmo original no funcione, por lo que necesitamos que nuestra entidad, o bot, aprenda a lidiar con ellos.

Un ejemplo de mundo variante puede ser el propio mundo de StarCraft [14], un videojuego de estrategia en el que dispondremos de una base y una serie de unidades que debemos controlar con el objetivo de destruir la base enemiga. En un entorno así, a veces es difícil gestionar las diferentes situaciones para conseguir ganar, por ello, este proyecto tiene como objetivo construir un bot que aprenda a controlar varias unidades y a lidiar con las diferentes situaciones que aparezcan, permitiendo descubrir estrategias para ganar a partir de la experiencia del propio bot.

Dado que no siempre sabemos en qué situación nos vamos a encontrar, parece idóneo utilizar el aprendizaje por refuerzo, que permite aprender de un entorno del que inicialmente no se tienen datos a través de la experiencia utilizando como medio un sistema de compensación de las diferentes acciones tomadas en función de sus consecuencias.

Chapter 1. Introduction

1.1 Motivation

Our main motivation when we chose this topic as main subject of our final degree project was the fact that all members of the group had played the videogame StarCraft in our childhood. All of us had studied the subjects about artificial intelligence and we had intention to study machine learning too, our attraction to these fields made us interested in this project from the beginning.

Everybody who has played any strategy game as Starcraft, has played games against the AI of the game, an opponent controlled by the computer which level of difficulty can be changed. These kinds of AI use to follow some patterns, and are managed by an algorithm against which it can be learned to play. But, what if it were not this way?, what if the opponent could learn from the games it plays and change its patterns to achieve victory?.

This is the idea that led us to choose this project, even though we knew we would not achieve that goal, we wanted to experiment how an AI could learn in this videogame.

When working in our project, we rely on the last year project, which left many possibilities opened to work in this field and deepen it.

1.2 Goals

From the beginning, we had clear that the main goal of this project was the automatic control and learning of many units coordinately and simultaneously, without a human component being the tutor, controlling or guiding them.

To reach that goal, we started contriving the way of manage just one unit, proposing small goals that will lead us to improve the development of our software and algorithms and so make the subsequent job easier. Eventually we were achieving goals and noticing that, meanwhile some objectives were easy to solve, other objectives weren't so easy as they seemed. Roughly, these are the goals that we have proposed:

- **Understanding the Q-Learning algorithm** to apply it in the learning of our AI. This algorithm has many variants and we needed to know which of them result useful in our tasks. We implemented this algorithm in a simple maze implemented in Java, and later we tested it in a maze created in the videogame, with the help of a unit that solved it.
- Allow a unit to learn to **destroy a structure**. Once the maze was solved, the next goal was to destroy an enemy turret. The main complexity of this problem was that our unit has to learn to shoot the target continuously, because if the unit stopped shooting and started to walk around the map, the unit would die.

- **Destroy other units.** The added difficulty is that in this case, our target can move and change its position. First, we were able to to destroy an enemy unit, and then we evolved our goals to stages where our unit had to move to survive. When the unit moved, it could reach a zone where it could be health.
- **Control many units at the same time.** Finally, we tried to use the learning of the AI to control actions of many units in combat.

1.3 Structure of the document

This document has all the information relevant to our work. The document is structured so the reader goes slowly immersing in the field we attend. Following we'll show a brief description of the contents of this document:

- **Chapter 2 - Background:** In this chapter we'll make an approach to StarCraft: Brood War, introducing some elements of the game that we will use, as well as the different features of reinforcement learning, their peculiarities against other kinds of learning, and an explanation of the algorithms we use; allowing the reader to familiarize with our project.
- **Chapter 3 - First experiment:** In this chapter we'll see the application of reinforcement learning over a first and simple example, and we'll analyze the obtained results.
- **Chapter 4 - Framework Q-learning:** In this chapter we'll explain how the framework that we developed works combining reinforcement learning and BWAPI.
- **Chapter 5 - Reinforcement learning in StarCraft:** In this chapter we'll tackle the different problems that we made to apply reinforcement learning in StarCraft. We'll also make an analysis of the obtained results and of the viability of the use of reinforcement learning in the problems.
- **Chapter 6 - Conclusions:** In this chapter we'll show our conclusions about our study of reinforcement learning and propose new open lines for future work.

Capítulo 1 Introducción

1.1 Motivación

La principal motivación que tuvimos a la hora de elegir este tema para nuestro trabajo de fin de grado fue el hecho de que todos los integrantes del grupo habíamos jugado al videojuego StarCraft en nuestra infancia. Además todos habíamos cursado las asignaturas de inteligencia artificial y teníamos intención de cursar la asignatura de aprendizaje automático, nuestro interés en estos campos nos hizo interesarnos en el proyecto desde el principio.

Toda persona que haya probado algún juego de estrategia como Starcraft, habrá jugado partidas contra una IA, un contrincante manejado por el ordenador al que se le puede cambiar el nivel de dificultad. Estas IA suelen seguir un patrón. Se manejan mediante un algoritmo contra el que se puede aprender a jugar. Pero, ¿y si no fuese así? ¿y si el contrincante contra el que nos enfrentamos pudiese aprender de las partidas que juega y cambiar su forma de actuar para conseguir la victoria?

Esta idea nos llevó a elegir este proyecto y, aunque esa meta era algo que sabíamos que no conseguiríamos, quisimos adentrarnos a probar cómo una IA podría aprender en este videojuego.

A la hora de encaminar nuestro proyecto, nos basamos en el trabajo realizado el año anterior, el cual dejaba abiertas muchas posibilidades para seguir trabajando en este campo y profundizar en él.

1.2 Objetivos

Desde el principio, hemos tenido claro que el objetivo de este proyecto ha sido el control y aprendizaje de manera automática de varias unidades de manera coordinada y simultánea, sin que ningún componente humano, haga de tutor sobre ellas o las controle o guíe de ninguna manera.

Para alcanzar ese objetivo, comenzamos planteándonos la manera de manejar a una sola unidad, proponiendo pequeños objetivos que nos llevaran a mejorar el desarrollo de nuestro software y algoritmos y así facilitar el trabajo posterior. Poco a poco fuimos cumpliendo los objetivos y nos fuimos dando cuenta de que, mientras que unos se resolvían de una manera sencilla, otros no eran tan simples como parecían a simple vista. A grandes rasgos, los objetivos en orden que propusimos fueron:

- **Entender el algoritmo Q-Learning** para aplicarlo en el aprendizaje de nuestra IA. Este algoritmo tienes muchas variantes y necesitábamos saber cuales nos serían útiles. Implementamos este algoritmo en un laberinto muy simple creado en java, y luego lo intentamos en un laberinto del videojuego con una unidad que lo resolviese.

- Hacer que una unidad **aprenda a destruir una estructura**. Una vez resolvimos el laberinto, el siguiente objetivo fue el de eliminar una torreta enemiga. La complejidad de este problema fue que nuestra unidad aprendiese a disparar continuamente, ya que si paraba de disparar para moverse por el mapa moriría.
- **Conseguir que una unidad destruya a otra/s.** La dificultada añadida es que en este caso, lo que queremos matar se mueve. Primero conseguimos destruir una unidad enemiga y avanzamos hasta pruebas donde nuestra unidad tenía que moverse. Al moverse, llegaba a una zona donde se podía curar, y así poder eliminar varios enemigos sin morir.
- **Controlar varias unidades a la vez**. Finalmente, intentamos utilizar lo aprendido por la IA para controlar las acciones de varias unidades en combate.

1.3 Estructura del documento

Este documento tiene toda la información relevante a nuestro trabajo.

El documento está estructurado de manera que el lector se vaya sumergiendo poco a poco en el campo que tratamos. A continuación mostraremos una breve descripción de los contenidos de este documento:

- **Capítulo 2 - Antecedentes:** En este capítulo haremos una aproximación al videojuego Starcraft: Brood War que utilizamos, explicando algunos de los elementos que usaremos del juego; así como también a las diferentes características del aprendizaje por refuerzo, sus peculiaridades con respecto a otros tipos de aprendizaje, y una explicación de los algoritmos que utilizamos; todo para permitir al lector familiarizarse mejor con nuestro trabajo.
- **Capítulo 3 - Primer experimento:** En este capítulo veremos la aplicación del aprendizaje por refuerzo sobre un primer ejemplo sencillo y analizaremos los resultados obtenidos.
- **Capítulo 4 - Framework Q-learning:** En este capítulo haremos una explicación del funcionamiento del framework que hemos desarrollado para compaginar el aprendizaje por refuerzo con BWAPI.
- **Capítulo 5 - Aprendizaje por refuerzo en StarCraft:** En este capítulo abordaremos los diferentes problemas que hemos realizado a la hora aplicar aprendizaje por refuerzo en el videojuego Starcraft. También haremos un análisis de los resultados obtenidos y de la viabilidad del uso del aprendizaje por refuerzo en el problema.
- **Capítulo 6 - Conclusiones:** En este capítulo mostraremos las conclusiones sacadas de nuestro estudio del aprendizaje por refuerzo y propondremos líneas abiertas para futuros trabajos

Capítulo 2. Antecedentes

2.1 Starcraft: Brood War

StarCraft: Brood War [14] es un videojuego de Estrategia en Tiempo Real (RTS, *Real Time Strategy*) desarrollado por **Blizzard Entertainment** en el cuál tendremos que elegir una de entre tres razas, cada una de ellas con una serie de características que la diferencian del resto y marcarán la estrategia a utilizar por el usuario. El objetivo del juego consiste en destruir a todos los jugadores enemigos.

El paradigma del juego es muy simple. Al empezar la partida, nuestra raza poseerá una base sencilla, pero con el paso del tiempo tendremos que ir creando diferentes tipos de unidades y edificios, y desarrollando diferentes mejoras. Para ello tendremos que recolectar diferentes tipos de recursos que tendremos que gastar en nuestros propósitos. Nosotros nos centraremos solamente en el aspecto combativo del juego, dejando de lado los aspectos relacionados con recolección de recursos o mantenimiento o construcción de la base.

En los siguientes apartados haremos una pequeña introducción a las características del juego para que el lector que nunca ha jugado se familiarice más con nuestro trabajo.

2.1.1 Unidades y acciones

En el juego nos encontraremos tres razas diferentes con las cuales jugar, cada una con unas características diferentes:

- **Terran**: La raza humana. Sus unidades son humanos o robots controlados por humanos. Se adaptan fácilmente a las adversidades.
- **Zerg**: Una raza alienígena que busca la perfección genética a cambio de asimilar otras razas. Se asemejan a un enjambre de insectos.
- **Protoss**: Una raza alienígena que busca acabar con la amenaza Zerg a cualquier coste. Son una raza humanoide futurista con poderes psiónicos.

Las unidades pueden realizar diferentes **acciones** como moverse, patrullar una zona o atacar. En el caso de los recolectores tenemos también las acciones de recolectar y la de construir. Algunas unidades tienen acciones especializadas con diferentes fines, por ejemplo, algunas unidades Terran tienen la capacidad de hacerse invisibles y algunas unidades Zerg pueden enterrarse para no ser vistas.

A la hora de desarrollar nuestros experimentos, desarrollaremos una serie de acciones más amplias de más alto nivel derivadas de las básicas anteriormente mencionadas. Algunos ejemplos de estas acciones pueden ser acercarse a aliados o atacar a enemigos con menos vida.

Cuando utilicemos como agente un grupo de unidades, utilizaremos acciones “de grupo”, las cuales nos permitirán lanzar órdenes al grupo entero, por ejemplo mover todos a la derecha, o atacar todos al blanco.

Unidades que utilizamos en el juego

Figura 1 – Marine Terran

Figura 2 – Médico Terran

Figura 3 - Zergling

Para nuestros experimentos, generalmente utilizaremos la raza Terran, en concreto la unidad Terran conocida como **Marine Terran** [Figura 1]. Se trata de su unidad básica de ataque a distancia.

Para los casos en los que el marine sea débil contra su contrincante, o contra varios contrincantes, utilizaremos al **Médico Terran** [Figura 2], capaz de curar a otra unidad aliada cuerpo a cuerpo. El objetivo de introducir a esta unidad será que el marine aprenda a utilizarla para curarse, ya sea poniéndose en su rango de visión para que el médico acuda a él, o directamente poniéndose a su lado para que le cure. Esta unidad será totalmente de apoyo, es decir, no será parte de nuestro agente, sino que será controlada por la IA del juego.

Como contrincante usaremos generalmente a la raza Zerg, normalmente nos decantaremos por la unidad de ataque cuerpo a cuerpo llamada **zerling** [Figura 3], ya que es ligeramente más fuerte que el marine (en una batalla normal 1 vs 1 ganaría el zerling), lo que permite buscar estrategias para evitar la muerte del marine más allá de quedarse quieto atacando.

Estructuras que utilizamos en el juego

Figura 4 – Centro de mando Terran

Figura 5 – Colonia de esporas

Figura 6 – Colonia hundida

En cuanto a estructuras utilizadas, para el caso de los Terran solamente utilizamos el **centro de mando** [Figura 4] a modo de baliza para la resolución de laberintos.

De las estructuras Zerg utilizamos dos estructuras: una básica llamada **colonia de esporas** [Figura 5] que simplemente se mantiene estática y otra llamada **colonia hundida** [Figura 6] la cual ataca a nuestro marine, y empezamos a usar cuando implementamos las acciones de atacar.

2.1.2 Map editor

El propio StarCraft viene con un **editor y creador de mapas**, el cual nos ha permitido crear los diferentes escenarios sobre los que probar nuestros algoritmos y experimentos. En este editor podemos desde crear diferentes terrenos y escenarios, o modificar las características de las unidades, hasta crear "eventos" que se ejecutarán dependiendo de ciertas condiciones del juego.

En la siguiente imagen [Figura 7] podemos observar la interfaz del creador de mapas. En la parte central podemos ver la representación directa de cómo quedará el mapa en la zona visible por el jugador. A la izquierda tenemos un catálogo de diferentes edificios y unidades de cada jugador, así como un minimapa representando el mapa completo. En la parte superior tenemos los diferentes menús que nos permitirán modificar las características tanto de los jugadores como de las unidades, así como gestionar los eventos anteriormente mencionados.

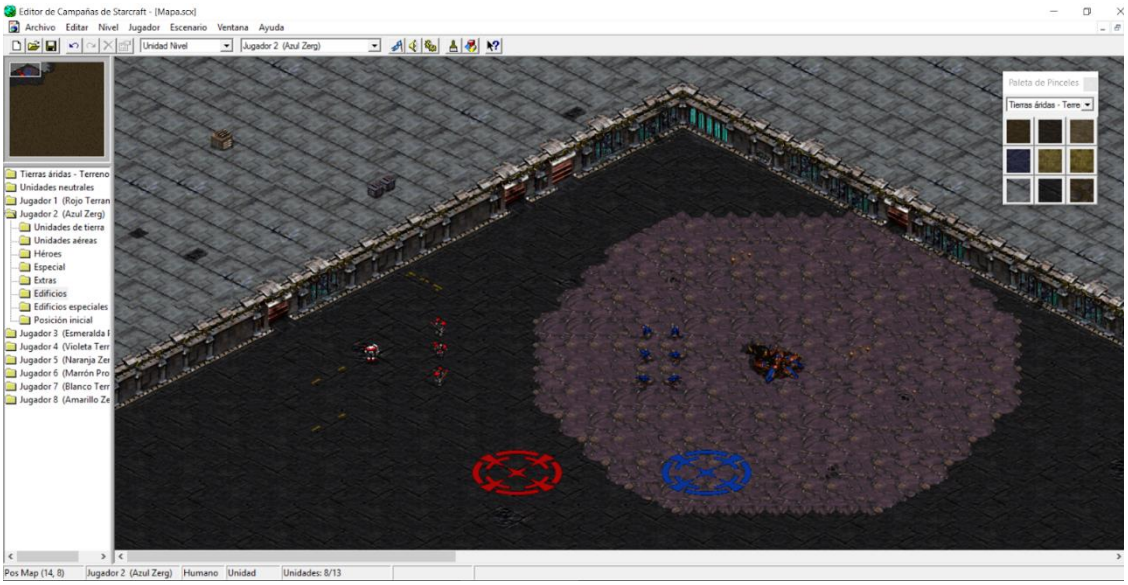


Figura 7 – Editor de campañas de StarCraft

2.1.3 BWAPI

BWAPI (Brood War API) [3] es una API que permite controlar las variables del videojuego StarCraft en tiempo de ejecución, lo que da libertad al jugador para

automatizar procesos dentro del juego. Inicialmente está creada para programar en C++, pero mediante el adaptador llamado **BWMirror** [4], puede ser utilizada también en Java.

A la hora de empezar con el proyecto, nos encontramos con el dilema de cuál de los dos lenguajes utilizar, decantándonos finalmente por Java, ya que nos resulta más fácil y rápida la programación en este lenguaje.

2.2 Aprendizaje automático

El **aprendizaje automático** [11] es una rama de **la inteligencia artificial** cuyo principal objetivo es conseguir que las máquinas aprendan. Hay varios enfoques dentro del campo del aprendizaje automático, algunos de ellos utilizan métodos matemáticos para aprender patrones sobre grandes sets de datos, como hacen los métodos de regresión o varios métodos de clustering, mientras que otros se basan en modelos y teorías biológicas como las redes neuronales, los algoritmos genéticos o el aprendizaje por refuerzo.

El aprendizaje automático está relacionado con multitud de campos como la estadística, la lógica, la biología o la psicología. El avance en todos estos campos produce también diversos avances, como nuevas teorías o aproximaciones en el campo del aprendizaje automático.

Existen 3 formas de aprendizaje automático, cada una de las cuales explicaremos a continuación en los siguientes apartados.

2.2.1 Aprendizaje supervisado

El **aprendizaje supervisado** utiliza como objeto de aprendizaje un conjunto de ejemplos etiquetados provistos por un supervisor externo. Cada ejemplo dado es una situación del problema acompañada por su etiqueta, la cual se corresponde con la acción correcta que debe tomarse, o la categoría a la que pertenece. El fin de este tipo de aprendizaje es poder generalizar las respuestas tomadas por el agente de manera que puedan ser correctas para situaciones no presentes en el set de entrenamiento. Este tipo de aprendizaje puede no ser práctico en problemas interactivos, ya que puede ser difícil obtener los ejemplos etiquetados y el agente tendrá que aprender de su propia experiencia.

2.2.2 Aprendizaje no supervisado

El **aprendizaje no supervisado**, trata típicamente de buscar estructuras ocultas en conjuntos de datos no etiquetados. Suelen extraer datos de sus entradas como características o correlaciones para así permitir agruparlos o relacionarlos entre sí. Su objetivo es poder aprender de sus entradas para finalmente poder clasificar nuevas entradas en los grupos que ya ha detectado A pesar de que puede parecer que el aprendizaje por refuerzo se puede englobar dentro del aprendizaje no supervisado ya que no utiliza ejemplos dados por un supervisor, el aprendizaje por refuerzo no trata de

buscar estructuras ocultas en un conjunto de datos, sino que trata de maximizar (o minimizar) una señal de recompensa.

2.2.3 Aprendizaje por refuerzo

El **aprendizaje por refuerzo** [10] es una forma de aprendizaje automático inspirado en la **psicología conductista**. Utiliza como base el proceso de aprendizaje del **condicionamiento operante**, según el cual, el comportamiento de un individuo se fortalece por las consecuencias producidas por ese comportamiento, razón por la que a esas consecuencias se las llama **reforzadores**. Aplicando esta visión conductista del comportamiento, asociaremos una acción con una recompensa positiva (**premio**) en el caso de querer que se repita, o con una recompensa negativa (**castigo**) en el caso de no querer que se repita.

Desde el punto de vista computacional, en este tipo de aprendizaje pretendemos que un agente explore y explote posibles caminos que llevan a la solución a un problema (**medio**) para obtener una recompensa (**fin**), la cual se pretende maximizar (o minimizar) y puede ser positiva si el resultado al problema es el deseado, o negativa en el caso de que sea un resultado no deseado o de que el camino para hallar ese resultado no sea el adecuado o pueda ser mejorable.

La principal característica de este tipo de aprendizaje que lo diferencia de los otros dos, es que no existe un conjunto inicial de datos, ya sean etiquetados o no, sobre los que aprender. En este caso, el agente aprende interactuando directamente con el **entorno**, lo que le da la capacidad de adaptarse dinámicamente a los cambios que puedan acaecer en el.

Como se puede observar en el siguiente diagrama [Figura 8], el funcionamiento del agente es bastante simple. El agente toma una acción A_t observando la información disponible en su entorno, y como resultado de tomar la acción, obtiene cierta recompensa R_{t+1} dependiente de este nuevo estado y la utiliza para actualizar su conocimiento, a continuación actualiza su estado S_t al estado actual S_{t+1}.

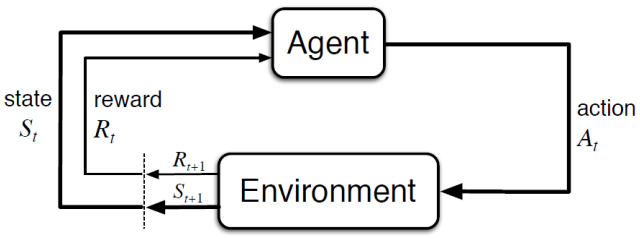


Figura 8 – Diagrama de aprendizaje por refuerzo

2.3 El algoritmo Q-Learning

El **algoritmo Q-learning** [10] será el algoritmo que usaremos para el aprendizaje de nuestro bot. Se basa en las ideas del **TDL (Temporal-Difference Learning)**. TDL surge de una combinación de las ideas de los métodos de Monte Carlo, gracias a los cuales puede aprender directamente de la experiencia sin tener un modelo completo de la dinámica del entorno sobre el que aprende, y de las ideas de la programación dinámica, según el cual actualiza los nuevos valores aprendidos en base a valores aprendidos anteriormente sin esperar a que haya un resultado final a nivel global. La relación entre TDL, programación dinámica y los métodos de Monte Carlo suele darse frecuentemente en la teoría del aprendizaje por refuerzo.

El algoritmo almacena la información en una **tabla de valores** que consta de una serie de dimensiones determinadas por el número de características que conoce el agente de su entorno, es decir, su estado. Por ejemplo, si tenemos un agente que resuelve laberintos, el agente conocerá su posición en el mismo, por lo que sería natural que la primera dimensión fuera la coordenada x en la que se encuentra, y la segunda dimensión la coordenada y. Por cada estado, el agente podrá tomar una acción, que es lo que queremos reforzar, por lo que hay una dimensión extra que especifica la acción a tomar. Cada celda de la tabla tendrá un valor que será más alto cuanto más preferible sea ese movimiento. Los valores de la tabla se van actualizando en función de la cantidad de recompensa recibida al ejecutar acciones en su entorno, reforzando así secuencias de acciones que permiten al agente obtener mejores recompensas.

Una buena aproximación para entender el algoritmo puede ser verlo como un **autómata de estados** [Figura 9] en el cual, cuando nos encontramos en un estado, lo que vamos a reforzar será una transición a otro estado. En el ejemplo del laberinto, para una posición dada, nuestro agente puede tomar 4 acciones (algunas de ellas posibles y otras no) que le llevarán a otro estado. El siguiente autómata representa un laberinto sin paredes en el cual hay que ir desde el estado (0,0) al estado (2, 2).

En este ejemplo, tendremos una tabla de tres dimensiones. La primera y la segunda dimensión se corresponderán con las coordenadas x e y en las que se encuentre el agente, ambas serán de tamaño 3, mientras que la tercera dimensión será de tamaño 4, consiguiendo un total de 36 celdas en la tabla (Algunas de ellas no posibles, como ir a la izquierda en (0, 0).

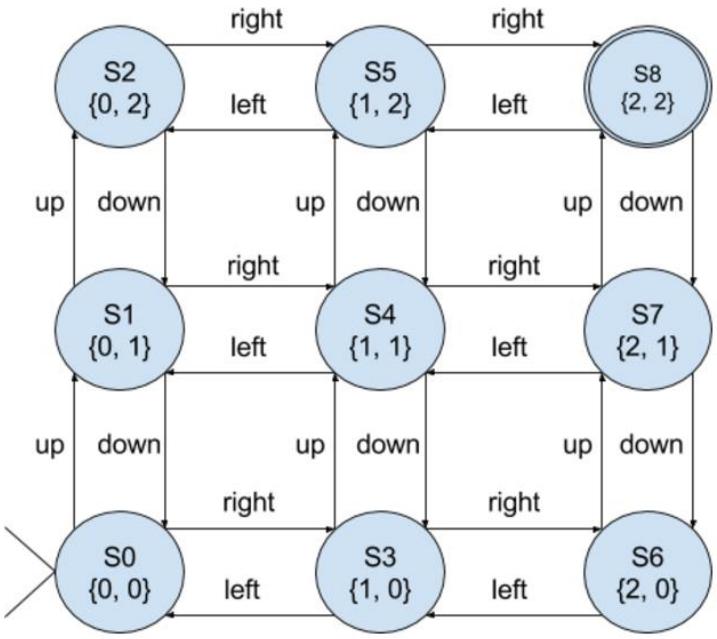


Figura 9 – Representación de estados de laberinto y transiciones como autómata

En su forma más simple, el Q-Learning de un solo paso, está definido como:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha * [R_{t+1} + \gamma * \max(Q_{t+1}, a) - Q(S_t, R_t)], 0 \leq \alpha < 1, 0 \leq \gamma < 1$$

- **S_t:** Estado concreto del bot en el entorno.
- **A_t:** Acción tomada por el bot
- **$Q(S_t, A_t)$:** Tabla en la cual se guardan los valores. Por cada par estado-acción, guarda un valor que es mayor cuanto más preferible sea tomar la acción A_t en el estado S_t.
- **R_{t+1}:** Recompensa obtenida en el estado S_{t+1}, al cual llegamos al tomar la acción A_t en el estado S_t.
- **α:** Parámetro del algoritmo que indica cuánto aprenderemos. Cuanto mayor sea alpha, si estamos en un estado S_t en el que queremos reforzar la acción A_t, mayor será el incremento del valor $Q(S_t, A_t)$ en la tabla.
- **γ:** Parámetro del algoritmo que indica cuánto se propagará lo aprendido en un estado S_{t+1} a los estados S_t con una acción A_t que lleve al estado siguiente S_{t+1}.

En esta definición, actualizamos cada par estado-acción a partir de su propio valor, y un valor limitado por alpha que depende del valor del siguiente estado óptimo (Limitado por gamma) y la recompensa obtenida en el siguiente estado. A partir de ello, podemos diseñar el algoritmo Q-Learning de un solo paso como sigue:

- Inicializamos arbitrariamente la tabla Q para todos los estados s ε S y para todas las acciones a posibles en cada estado S.
- Inicializamos los estados finales de Q a 0.
- Para cada episodio
    - Inicializamos S a estado inicial.
    - Para cada paso del episodio, hasta que S sea terminal
        - Elegimos una acción A partiendo del estado S utilizando una política derivada de Q (Ej de política: ε-greedy).
        - Tomamos acción A y guardamos el estado resultante en S'.
        - Obtenemos el valor de la recompensa para el estado S'.
        - Actualizamos el valor de la tabla Q utilizando los valores obtenidos con la fórmula anterior.
        - Actualizamos S, asignándole el estado siguiente S'.

Obtenemos entonces el siguiente algoritmo en pseudocódigo:

```
Inicializar arbitrariamente Q(S, A), ∀s ε S.  a ε A(s)
Q(S_terminales, -) = 0
Repetir (para cada episodio)
      Inicializar S;
      Repetir (para cada paso del episodio)
            Elegir A para aplicar en S usando una política derivada de Q
(ej, ε-greedy)
            Tomar acción A y guardar estado resultante S', observar R de S'
            Q(S, A) = Q(S, A) + α ·[R + γ ·max_a(Q(S, A)) - Q (S_t, A_t)]
            S = S';
Hasta S terminal
```

2.3.1 Equilibrio exploración/explotación

Uno de los principales retos en el aprendizaje por refuerzo no presente en otros tipos de aprendizaje, es especificar la relación entre explotación y exploración. La **Exploración** es el hecho de explorar los diferentes caminos o soluciones que puede haber para solucionar un problema, mientras que **explotación** es el uso de los caminos o soluciones exploradas. El agente va a intentar realizar diferentes acciones ante determinadas situaciones, y progresivamente favorecerá las que parecen ser mejores. El problema al especificar la relación consiste en decidir con qué frecuencia explorar nuevos caminos en lugar de explotar los ya conocidos, de modo que no se falle en la resolución del problema.

Para resolver este problema, podemos utilizar varias políticas o estrategias. Nosotros utilizaremos aquí la estrategia **ε-greedy**, la cual consiste en utilizar una estrategia voraz que elige la mejor acción que se puede tomar por cada estado (explotación), excepto en un porcentaje de casos determinado por el parámetro ε, en los cuales se elegirá una acción aleatoria de entre las posibles (Exploración). La principal ventaja de esta técnica consiste en que se puede fijar el número de accione exploratorias de una forma más o

menos aleatoria, de manera que será posible la exploración en cualquier momento. Mostraremos a continuación el algoritmo de la política ε-greedy:

```
Elegimos un número rand aleatorio entre 0 y 1
si (rand < ε)
      Elegimos a ε A(s) aleatoria
Si no
      Elegimos la mejor acción a ε A(s) posible
```

2.3.2 Explotación de los valores aprendidos

Si ejecutamos el algoritmo sobre un problema, actualizará la tabla Q(S, A) que será la que usaremos para tomar siempre las acciones aprendidas. Para esto, podemos hacer dos cosas. La primera de ellas, y más obvia, sería **modificar el algoritmo** original para que no tome acciones aleatorias y no modifique la tabla de manera que no modifique lo ya aprendido, quedando el siguiente algoritmo:

```
Repetir (para cada episodio)
      Inicializar S;
      Repetir (para cada paso del episodio)
            Elegir el mejor A para aplicar en S
            Tomar acción A y guardar estado resultante S'
            S = S';
Hasta S terminal
```

Esta opción implicaría tener un algoritmo para aprender y otro diferente para utilizar lo aprendido. Otra opción, que permitiría tener un solo algoritmo para las dos tareas, sin tener que modificar el algoritmo original, y que permitiría alternar entre una tarea y otra de forma sencilla, consistiría en **asignar el valor 1 al parámetro ε** de ε-greedy de manera que siempre se tomen las acciones aprendidas y nunca aleatorias, y el valor 0 al parámetro α del algoritmo, de forma que no se actualicen los valores de la tabla y no se modifique lo aprendido.

2.3.3 Función de temperatura

Hemos visto dos opciones sencillas para, una vez hemos aprendido, utilizar estos valores de manera que no se modifiquen y no se tomen aleatorios. Estos métodos implican que a partir de cierto momento, el algoritmo deberá cambiar drásticamente para adecuarse a la nueva tarea, pero podemos automatizar el proceso de manera que a medida que avance el algoritmo, los valores α y ε se vayan modificando progresivamente hasta llegar α a 0 y ε a 1.

Para conseguir este efecto, utilizaremos las siguientes funciones de temperatura que hemos desarrollado para este propósito:

$$\alpha = \alpha_{ini} - \alpha_{ini} * \left(e^{\frac{-T_\alpha}{(i+1)}} \right), \qquad 0 \leq \alpha < 1, \qquad 0 \leq \alpha_{ini} < 1, \qquad 0 < T_\alpha < \infty$$

$$\varepsilon = \varepsilon_{ini} + (1 - \varepsilon_{ini}) * \left(e^{\frac{-T_\varepsilon}{(i+1)}} \right), \qquad 0 \leq \varepsilon < 1, \qquad 0 \leq \varepsilon_{ini} < 1, \qquad 0 < T_\varepsilon < \infty$$

Donde α y ε son los valores actuales de los parámetros, α_{ini} y ε_{ini} son los valores iniciales de α y ε, y T_α y T_ε son dos parámetros que determinan la velocidad de crecimiento o decrecimiento de las funciones. La i es el número de la iteración actual del algoritmo.

La forma de utilizar estas funciones consiste en llamarlas en cada episodio del algoritmo. Los valores de T_α y T_ε hay que elegirlos según el problema a resolver, ya que dependiendo del número de iteraciones, la evolución del valor puede no ser la adecuada. El valor de T debe ser menor cuanto más rápido queramos que se aproxime.

Por ejemplo, si tenemos un épsilon igual a 0.9 y queremos que se aproxime lo más posible a 1 pasadas 200 iteraciones, si y T_ε vale 50, el valor de épsilon no llegará a 0.98 [Figura 10], sin embargo, si T_ε vale 5, se aproximaba bastante más, hasta llegar a 0,9976 [Figura 11].

Funciones de temperatura según el valor de T_ε

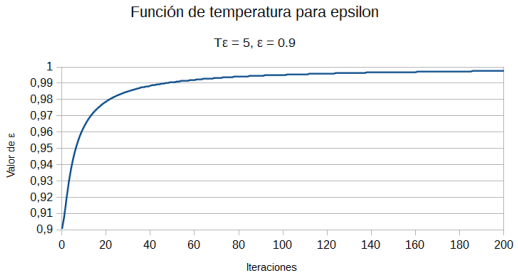


Figura 10
Función de temperatura para T_ε = 5

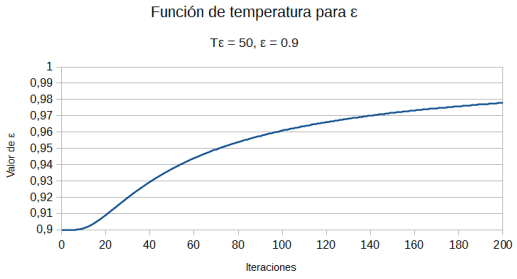


Figura 11
Función de temperatura para T_ε = 50

En el caso de alpha, si queremos que se aproxime a 0 tras 200 iteraciones ocurre lo mismo. Si utilizamos T_α = 5, lograremos una aproximación de 0.012 [Figura 12], pero utilizando T_α = 50, lograremos llegar hasta 0.11 [Figura 13].

Funciones de temperatura según el valor de T_α

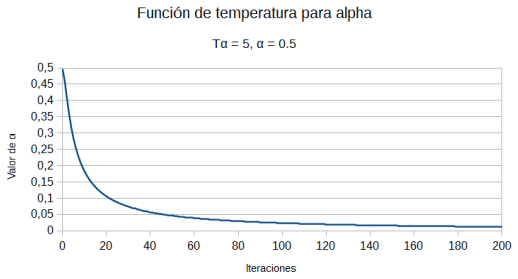


Figura 12
Función de temperatura para T_α = 5

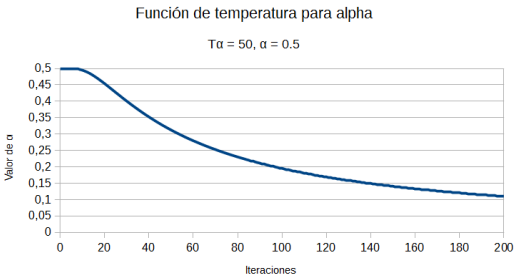


Figura 13
Función de temperatura para T_α = 50

2.4 El algoritmo Q-Learning(λ)

Cuando Chris Watkins propuso por primera vez el algoritmo Q-learning [16], también propuso una vía simple para combinarlo con las **trazas de elegibilidad**. Las trazas de elegibilidad son una forma de guardar temporalmente la ocurrencia de un evento, permitiendo diferenciar entre eventos e información de aprendizaje. Esto lo que nos lleva es, en última instancia, a una convergencia más rápida.

La idea básica de las trazas de elegibilidad es no sólo tener en cuenta la recompensa para el estado actual, sino para varios. Si lo vemos como ir hacia delante, sería tener en cuenta la recompensa de *n* pasos siguientes, mientras que si lo vemos como una "acumulación de recompensas", teniendo que mirar hacia atrás, la idea es afectar a *n* estados anteriores. En nuestro caso, para el Q-Learning de Watkins, tenemos que verlo como la segunda alternativa.

Para nuestra implementación, vamos a mantener una tabla de Trazas de Elegibilidad análoga a la Q, donde tenemos que Et(s, a) ≡ valor de la traza de elegibilidad en el instante *t*, para el estado *s* y la acción *a*. La actualización de la Q ahora depende de las trazas de elegibilidad de la siguiente forma:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha\delta_t E_t(s,a), \forall s \in S, a \in A$$

Como vemos, la Q ahora depende de las trazas de elegibilidad. El algoritmo completo en pseudocódigo se vería así:

```
Inicializar arbitrariamente Q(S, A), ∀s ε S.  a ε A(s)
Repetir(para cada episodio)
      E(S, A) = 0 ∀ s ε S.  a ε A(S)
      Inicializar S, A;
      Repetir(para cada paso del episodio)
            Tomar acción A y guardar estado resultante S', observar R de S'
            Elegir A' para S' usando política derivada de Q (ej, ε-greedy)
            A* = max_a Q(S', A) + 1;
            δ = R + γQ(S', A*) - Q(S, A)

            //Elegir una de las tres
            E(S, A) = E(S, A) + 1                (Acumular trazas)
            O E(S, A) = (1 - α)·E(S, A) + 1      (Dutch traces)
            O E(S, A) = 1                        (Reemplazar trazas)
            Para todo s ε S, a ε A(s)
                  Q(S, A) = Q(S, A) + α·δ·E(S, A)
                  Si A' = A* entonces hacer E(S, A) = γλE(S, A)
                  Si no hacer E(S, A) = 0
            S = S';
            A = A';
Hasta S terminal
```

Como vemos en el pseudocódigo, lo que vamos haciendo es "marcar" por donde vamos pasando, y actualizar los valores de la Q en base a esto. Por ejemplo, si no hemos

realizado la acción *A* en el estado *S,* el valor de E(a, s) será 0, y en la actualización de la Q (Q(S, A) = Q(S, A) + α·δ·E(s, a)), el valor que teníamos quedará sin modificar.

Es importante destacar lo enormemente ineficiente que es esta implementación, pues para todas las iteraciones hay que recorrer las dos tablas al completo.

Desafortunadamente, el hecho de que las trazas de elegibilidad tomen el valor 0 cuando se toma una acción exploratoria hace que se pierda en parte la ventaja de usarlas. Si las acciones de exploración son frecuentes, ya que a menudo aparecen tempranamente en el aprendizaje, entonces rara vez las copias de seguridad se realizan de más de uno o dos pasos, y el aprendizaje es un poco más rápido que en el caso del one-step Q-learning. Esto viene provocado porque en las exploraciones, donde ocurre que A' != A*, hace que E(s, a) = 0

2.5 Adversarios

Para poder estudiar mejor a las posibles adversidades que puede encontrar el agente y cómo puede aprender de ellas, creamos una pequeña aplicación utilizando el algoritmo Q-learning que aprendía estrategias para jugar al juego de 3 en raya (tic-tac-toe).

La recompensa que le damos inicialmente al agente, será una recompensa de 50 si ha habido empate o si el jugador ha bloqueado un movimiento enemigo ganador, 80 si el agente ha ganado, y una recompensa de -10 si el jugador ha perdido o si no ha bloqueado al jugador enemigo.

Una vez hecha la aplicación, nos planteamos entonces que tipos de adversarios puede tener el agente. Una primera idea, fue hacer que el agente aprendiera contra un jugador cuyos movimientos fueran aleatorios, lo que llevó a que el agente aprendiera a ganar ante muchas situaciones, pero de formas no óptimas, por lo que si ponemos al agente a jugar contra un jugador humano, a no ser que éste cometiera grandes equivocaciones, no era capaz de ganarle.

En la figura 14 se puede ver una sucesión de movimientos no óptima que ha aprendido el agente (azul) para ganar, lo que es debido no solo a que el adversario no bloquea el movimiento ganador del agente, sino también a que recompensamos las sucesiones de movimientos en las que el agente gana, sean o no óptimas. Por lo que decidimos bajar la recompensa que obtiene el agente al ganar a 50, y pensar en otros posibles adversarios.

Estrategias seguidas por el agente de 3 en raya

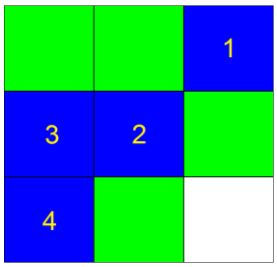


Figura 14 – Estrategia seguida contra un oponente aleatorio

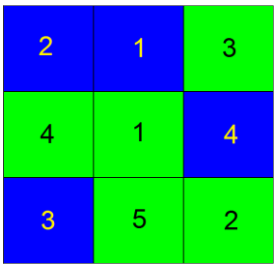


Figura 15 – Estrategia seguida contra un oponente humano

La siguiente opción que se nos ocurrió, fue poner al agente contra un jugador humano y que aprendiera contra él. Si el jugador es muy bueno, y nunca comete errores, el agente acabará aprendiendo a empatar continuamente, pero si de repente él comete algún error, no ganará porque nunca ha ganado ni ha aprendido a ganar. Utilizar un jugador humano no es una idea demasiado buena por el tiempo que tarda el humano en decidir los movimientos. En figura 15 podemos ver una sucesión de movimientos en la que el agente (azul) bloquea los posibles movimientos ganadores del humano (verde) llevando a un empate:

Si eventualmente durante el aprendizaje el jugador humano empieza a cometer fallos, el agente tendrá oportunidad de aprender a ganar. En el ejemplo del 3 en raya, podríamos automatizar esto utilizando un algoritmo minimax e-greedy con una función de temperatura para e, de manera que empezaremos tomando un 100% de decisiones óptimas, hasta tomar todas aleatorias.

Otra opción para que el agente aprendiera mejor, fue poner al algoritmo a jugar contra sí mismo, en un primer vistazo parecía que aprendía estrategias bastante mejores que contra un oponente aleatorio, pero aún aprendía algunas estrategias con movimientos no óptimos (ya que al principio los movimientos que toma son aleatorios), que, al igual que contra componentes aleatorios, al repetirlos tras aprenderlos los reforzaba y más adelante costaba corregirlos.

Finalmente concluimos que la mejor forma de aprender es contra un adversario que inicialmente juegue de una forma perfecta, de manera que el agente aprenda a no perder, a partir de ahí, el adversario debería ir fallando eventualmente para darle la oportunidad al agente de aprender a ganar, y más adelante empezar a fallar más frecuentemente para aprender a aprovecharse de jugadores que tengan estrategias malas. Habremos aprendido estrategias para no perder o estrategias ganadoras.

Capítulo 3. Primer experimento: Laberinto

Para ilustrar mejor las ideas anteriormente expuestas, mostraremos a continuación el primer experimento que hicimos con este tipo de aprendizaje. Para este primer experimento no hemos utilizado el videojuego StarCraft, por lo que el programa entero ha sido desarrollado por nosotros para comprender mejor el funcionamiento del algoritmo Q-learning.

Los parámetros utilizados para las siguientes pruebas son:

Parámetros del problema	
α	0.8
γ	0.9
ε	0.1
T_ε	1

3.1 Descripción

El problema consistirá en la resolución de un laberinto en el menor número de pasos. El laberinto [Figura 16] estará formado por una cuadrícula de 20x20 en la cual tendremos paredes (Negro), caminos (Blanco), una posición de inicio (Verde), una salida (Rojo) y un agente (Azul).

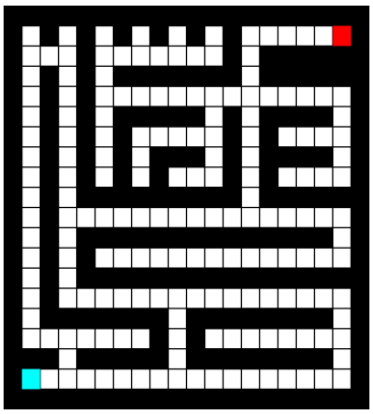


Figura 16 – Laberinto del ejemplo

El número de posibles estados es relativamente pequeño, ya que está acotado superiormente por el número de filas, el número de columnas y el número de acciones. La cota superior al número de estados es: 20*20*4 = 1600 posibles estados del agente. Este número se ve bastante reducido en el problema ya que hay bastantes estados imposibles, como los estados en los que el agente se encuentre sobre una de las paredes, y los estados en los que el agente intente moverse a una de las paredes.

Las acciones posibles que podrá tomar el agente en este problema son los movimientos básicos arriba, abajo, derecha e izquierda.

El objetivo del agente será alcanzar la salida. Le daremos recompensa positiva alta al llegar a la posición final (100), y una pequeña recompensa por cada paso que dé (0.1), para así reforzar los caminos que más se toman y acelerar el aprendizaje.

Por cada paso que dé, el agente actualizará la tabla Q(S, A), representando el estado S como un par (x, y), que indicará la casilla en la que se encuentra, y la acción como un enumerado. Si la acción tomada lleva a un estado que otorga recompensa positiva, aumentará el valor de Q(S, A), indicando que tomar la acción A en el estado S es algo conveniente.

En este problema encontramos 3 posibles soluciones directas que no llevan a callejones sin salida representadas en las figuras 17, 18 y 19.

Soluciones del laberinto

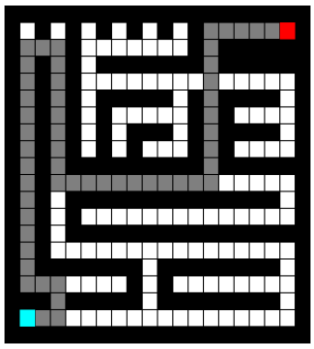


Figura 17 – Solución 1 de 54 pasos

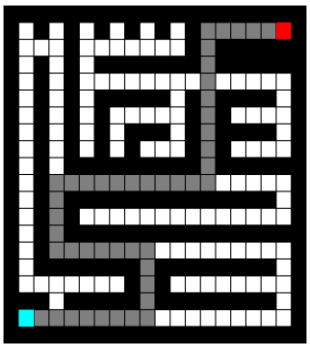


Figura 18 – Solución 2 de 46 pasos

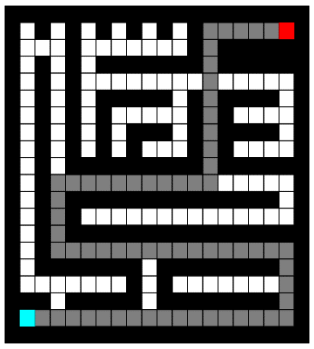


Figura 19 – Solución 3 de 64 pasos

3.2 Tabla Q

Si ejecutamos el agente sobre el laberinto por primera vez, todos los movimientos que tomará serán aleatorios al no haber obtenido una recompensa grande antes. En este primer episodio, el agente puede llegar a tomar un número de pasos total mayor a 10000.

En la figura 20 mostraremos un mapa de temperatura de la tabla Q(s, a), es decir, una representación de la tabla Q(s, a) sobre el laberinto, en la cual, cada celda tendrá 4 subceldas, una en cada dirección que puede tomar el agente desde esa celda. Cada subcelda tendrá un color diferente, en función del valor que contiene la tabla para tomar esa acción, siendo los colores más claros los valores más altos y los más oscuros los más bajos. Hemos aumentado la luminosidad y el contraste de la imagen para una visión más precisa de los caminos obtenidos.

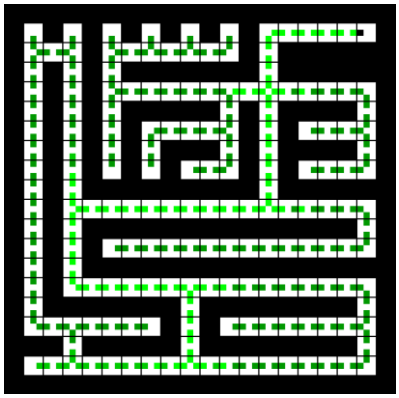


Figura 20 – Mapa de temperatura de la tabla del laberinto

3.3 Resultados

En la gráfica de la figura 21 se puede observar que el número de movimientos efectuados por el agente se reduce a medida que avanzan los episodio hasta aproximadamente el episodio 60 donde ya ha aprendido el camino óptimo para resolver el laberinto. Como podemos observar en la figura 22, a partir de este punto, el número de movimientos presenta pequeñas variaciones debidas a los movimientos aleatorios efectuados por la política e-greedy.

Gráficas de movimientos del laberinto

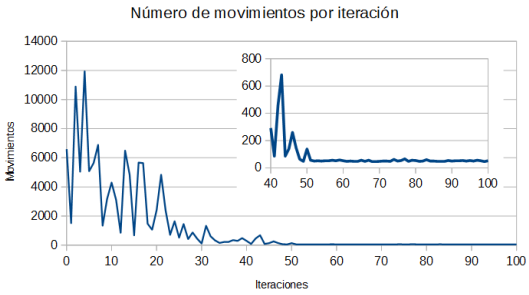


Figura 21 – Movimientos por iteración en el laberinto sin función de temperatura

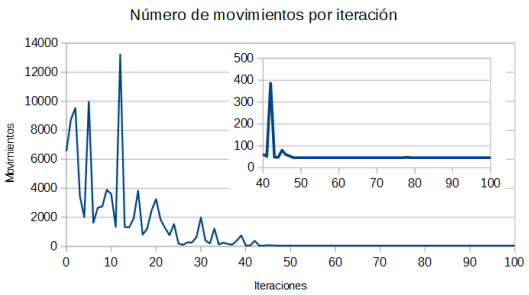


Figura 22 – Movimientos por iteración en el laberinto con función de temperatura

El problema de la exploración-explotación se puede ver aquí claramente debido a que aunque haya aprendido, el agente tomará eventualmente alguna acción aleatoria para favorecer la exploración de posibles nuevas soluciones. Si no reducimos la capacidad de explorar con el tiempo, no realizará el recorrido en el menor número de pasos, y en alguna ocasión dará pasos en falso hacia atrás o hacia otros caminos. Si eliminamos la opción de que tome acciones aleatorias una vez ha aprendido la solución óptima, siempre la tomará.

Por otro lado, si nos apresuramos al eliminar la capacidad de exploración, y el algoritmo ha aprendido una de las soluciones no óptimas, por ejemplo, la solución 3, estaremos evitando que el agente llegue siquiera a recorrer la solución óptima, por lo que nunca la

encontraremos y explotaremos soluciones alternativas diferentes a la óptima. Debemos tener cuidado al elegir el momento en el que el agente dejará de explorar, tarea en la que puede ayudarnos utilizar una función de temperatura como mencionamos en el apartado anterior.

Asignando la función de temperatura a épsilon nos permitirá automatizar el proceso de reducir el porcentaje de acciones aleatorias a tomar. Utilizando $T_\varepsilon = -1$, tras 60 iteraciones conseguiremos que épsilon tenga el valor 0.001, por lo que la probabilidad de que salga una acción aleatoria será muy pequeña. Como podemos ver en la figura 22, las variaciones en el número de movimiento a partir de las 60 iteraciones se han reducido drásticamente, por lo que prácticamente siempre consigue el mínimo número de movimientos a partir de ese momento.

Capítulo 4. Framework Q-Learning

Desgraciadamente, uno de los mayores desafíos en este proyecto ha sido trabajar con BWAPI, y no con el Q-Learning. Las razones han sido diversas; por un lado, era una API nueva para nosotros y hemos tenido que aprender a utilizarla desde 0 (Aunque esto pasa con cualquier API nueva). Por otro lado, la falta de documentación provoca que tengamos que hacer las cosas por ensayo y error, por lo que nos ha pasado en más de una ocasión que una cosa que aparentemente funciona, "de repente" deja de funcionar porque en realidad no funcionaba como nosotros pensábamos, pudiendo provocar efectos inesperados o hacer más cosas de las que pensábamos entre otras cosas.

Por todas estas razones, decidimos que lo más sensato es desarrollar nuestro propio **framework** sobre BWMirror. Este framework no sólo sería una capa de abstracción más sobre StarCraft, sino también un framework que nos ayude a centrarnos en lo más importante en lo que respecta al Q-Learning. Una gran ventaja de esto será que futuros trabajos podrán basarse sobre este framework, ahorrándose (parte) de los dolores de cabeza que nosotros hemos pasado.

A continuación presentamos la estructura de este framework, sus características, y los aspectos clave que permiten utilizarlo para realizar trabajos sobre BWAPI con Q-Learning.

4.1 Los elementos básicos: Bot, Agentes, Acciones y Toma de decisiones

En nuestro framework se presentan 4 elementos básicos diferenciados entre sí. Primero vamos a dar una breve descripción sobre ellos para tener una visión global. A lo largo de este capítulo los veremos más detalladamente.

- **Bot:** Es la capa que se comunica directamente con BWMirror. Recibe del agente master la señal para reiniciar la partida (y en última instancia comenzar una nueva iteración del Q-Learning) y envía al agente notificaciones sobre unidades destruidas.
- **Agente:** Veremos que se divide en Master, que es el agente que se comunica con el Bot; y el resto de Agentes, que ejecutan las acciones del motor de Toma de decisiones, les comunica las recompensas y el estado del entorno y notifican al master cuando se llega al estado final.
- **Acciones:** Hay dos tipos de acciones: las acciones que se comunican con BWAPI (por ejemplo, ordenando a las unidades que se muevan o ataquen), y las acciones que utiliza el motor de toma de decisiones, que básicamente son un enumerado que se pueden traducir directamente a las anteriores. Estas acciones son decididas por el Motor de toma de decisiones y ejecutadas por los agentes.

- **Toma de decisiones:** El motor de toma de decisiones o *Decision Maker* es básicamente la implementación del algoritmo de Q-Learning explicado previamente: en base al estado actual y a lo aprendido, decidir la siguiente acción a ejecutar. Las acciones se mandan a los agentes.

En la figura 23 se presenta un diagrama básico con las comunicaciones entre las partes:

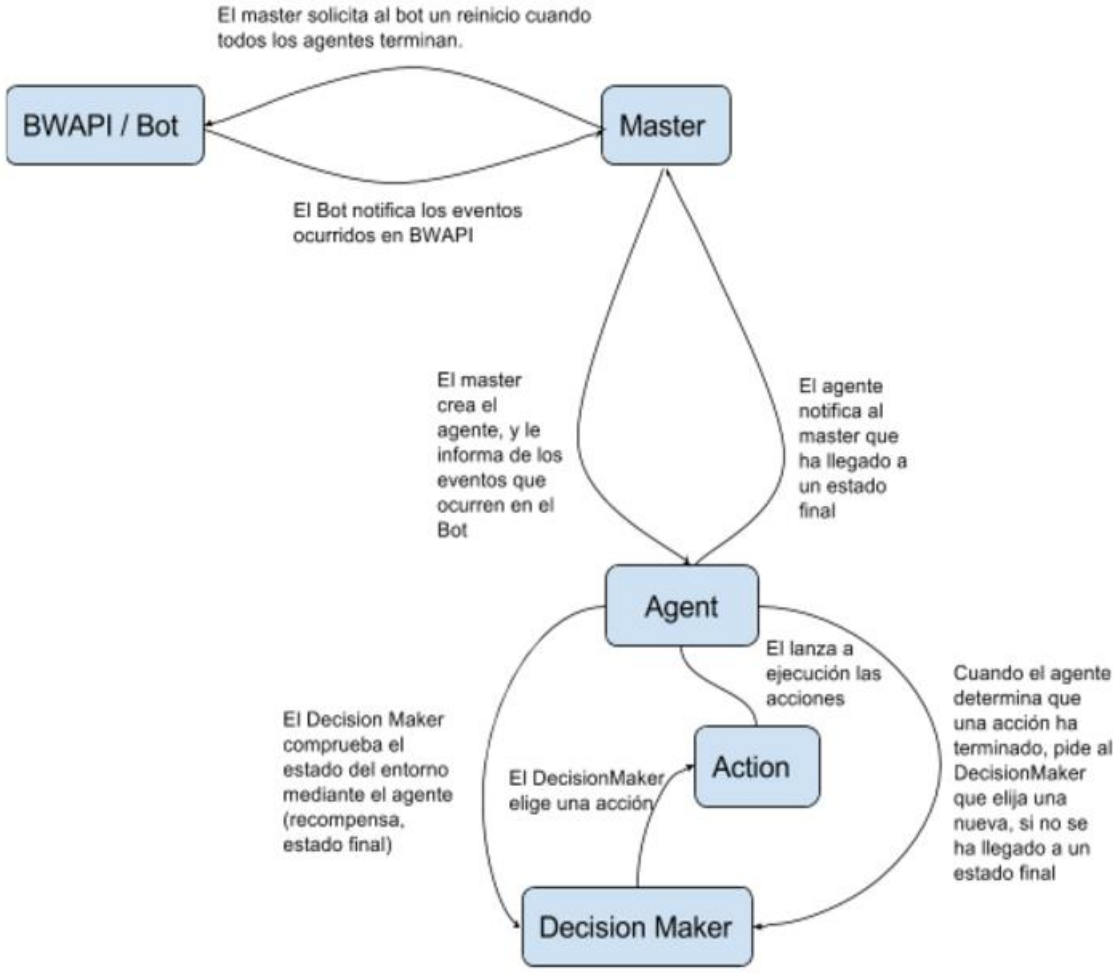


Figura 23 – Diagrama básico representando la comunicación entre las partes

Además de estos cuatro elementos básicos, también tenemos una GUI que veremos con detalle al final del capítulo.

4.2 Entendiendo BWAPI y aplicándolo al framework

Uno de los aspectos más importantes que hay que tener en cuenta para entender el funcionamiento del framework es el funcionamiento de BWAPI en sí.

El primer componente a tener en cuenta es la clase Bot, que extiende a la clase de BWMirror *DefaultBWListener,* con lo que conseguimos que nuestra clase Bot sea el punto de acceso entre nuestro framework y BWAPI (y en última instancia con StarCraft).

En Bot tenemos el método onFrame(), que sobrescribe a la implementación por defecto de *DefaultBWListener* (que no hace nada), método que es llamado en cada frame del juego. Junto a esto, con otro método que nos permite obtener todas las unidades de nuestro jugador/bot, tenemos prácticamente todo lo necesario para empezar a trabajar. En cada frame del juego, comprobamos el estado de todas nuestras unidades, para así poder realizar la toma de decisiones. A esto sólo le falta una cosa, y es saber cuándo una unidad ha muerto, tarea bastante sencilla, ya que existe un método onUnitDestroy(Unit unit), que es llamado cuando una unidad de nuestro jugador muere. Este método de *DefaultBWListener* también lo hemos sobrescrito en Bot.

Sin embargo, aquí se presenta un pequeño problema: este onUnitDestroy (...) se llama después del onFrame (por cada unidad destruida), como se presenta en la figura 24.

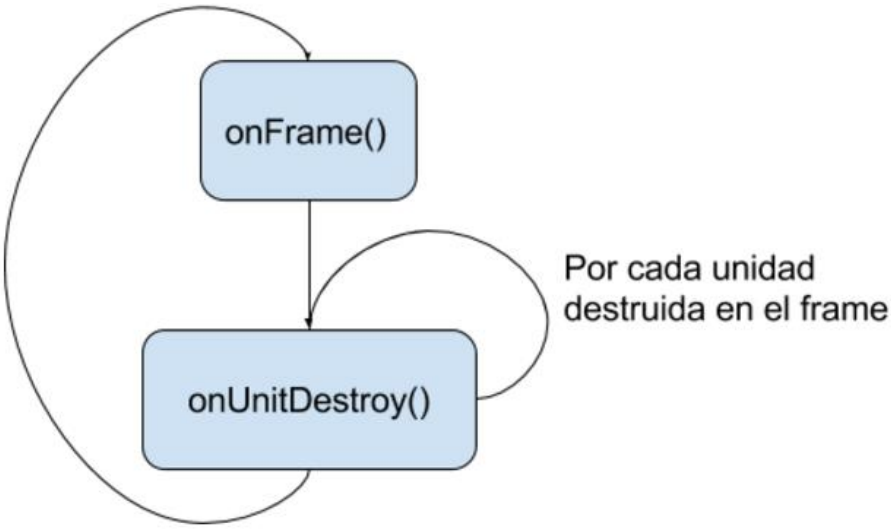


Figura 24 – Esquema de llamadas a onUnitDestroy()

Teniendo en cuenta este esquema, y viendo cómo estábamos planteando el Agente, decidimos que lo mejor sería realizar toda la comunicación entre Agentes y BWAPI de la siguiente manera:

- **Eventos:** Sirven para señalizar el fin de la iteración del Q-Learning. Cualquier agente podrá generar un evento que indique la terminación del agente provocado por llegar a un estado final. Cuando todos los agentes hayan finalizado, el master activará un flag en el Bot para que este reinicie la partida. Estos eventos podrían utilizarse para otros fines en posibles extensiones a nuestro framework.
- **Cola de acciones:** Sirve para comenzar la ejecución de acciones. Cada Agente tiene una cola de acciones que en cada **frame** (lo que viene a ser en cada llamada al método onFrame del agente) se **lanzan a ejecución**. Por otra parte, cuando el Decision Maker determina cual es la **siguiente acción** a ejecutar, la **añade** a esta cola. En el proyecto que nosotros hemos

desarrollado sólo puede haber una acción a la vez por cada agente, pero está dispuesto así por si en un futuro esto se modificase en trabajos posteriores.

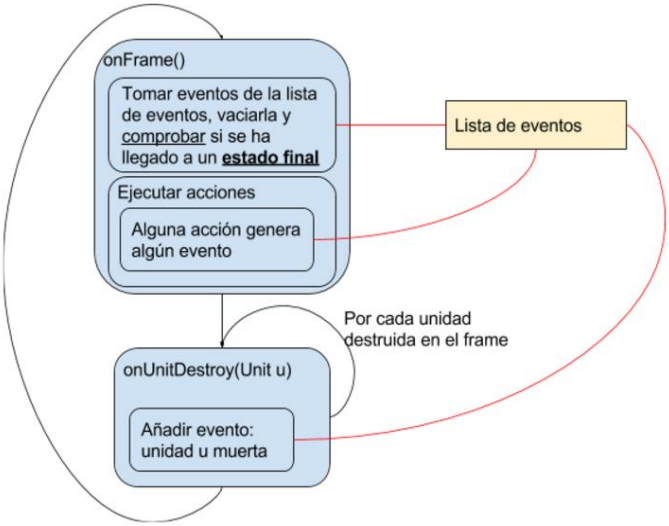


Figura 25 – Esquema de la estructura del Framework en las versiones iniciales

El esquema de la figura 25 representa la estructura del Framework. Como veremos más adelante, no representa exactamente la estructura del framework final (aunque sí de las versiones iniciales), pero representa la idea general de una manera bastante simple y comprensible.

4.3 Estructura de los Agentes y las Acciones

La estructura de los agentes es muy importante en nuestro framework. Gracias a ella podemos crear nuevos agentes sin demasiado esfuerzo, lo que permite centrarse más en el diseño de los Agentes y menos en programar.

Por otro lado, tenemos las acciones. Claramente van ligadas al agente que las ejecuta; por ejemplo un agente de alto nivel podrá tener unas acciones como centrarse en recolectar o centrarse en explorar el mapa, mientras que uno de más bajo nivel hará cosas como atacar a todas las unidades que tengo en rango, con cuidado de que no se me acerquen mucho. De la misma manera que con los agentes, queremos que la adicción de nuevas acciones sea algo sencillo y que sólo haya que preocuparse de la complejidad de la acción en sí.

4.3.1 Agente Master

Es importante remarcar la existencia de un agente especial, el master, que se encargará de la comunicación entre el Bot y el resto de agentes. Para cualquier experimento sólo existirá un master independientemente del tipo y número de agentes. En concreto, en los métodos que comentamos previamente, onFrame y onUnitDestroy, el Bot llama a

métodos homónimos del master; y el master a su vez se encargará de notificar al resto de agentes de manera pertinente.

Además, el resto de agentes podrá notificar al master ciertos eventos, como por ejemplo el fin del entrenamiento por llegar a un estado final. Cuando todos los agentes que controle el master lleguen a su fin, se dará la iteración por terminada, y se reiniciará el bot para dar paso a una nueva iteración.

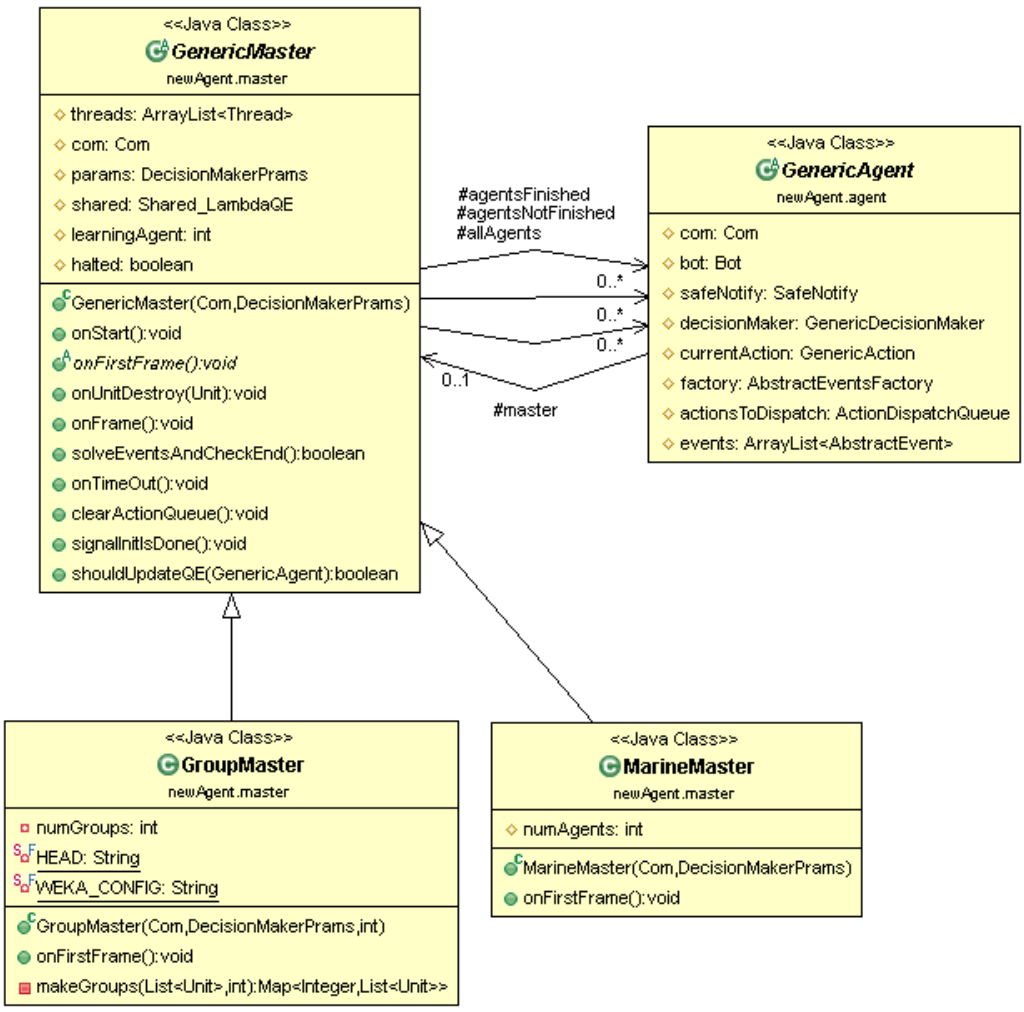


Figura 26 – Diagrama de clases de master y agente genérico

4.3.2 Resto de agentes

A parte del master, el resto de agentes se corresponden con la idea de Agente de Inteligencia artificial (en nuestro caso, del aprendizaje por refuerzo) que decide cuál es la acción a tomar. Como ya venimos adelantando, en nuestro framework contemplamos la posibilidad de tener varios agentes a la vez. Para que esto sea posible, por un lado mantenemos el Master explicado en el punto anterior, y por otro mantenemos una estructura del código adecuada para que sea sencillo extender los agentes, y por supuesto que ejecute correctamente las acciones decididas.

En el diagrama de clases de la figura 27, se presentan las clases de los agentes. Como se puede observar tenemos la clase genérica, de la que extiende la clase abstracta UnitAgent, para los agentes que controlan una única unidad (como MarineUnitAgent), y la clase MarineGrouAgent, que controla un grupo de marines.

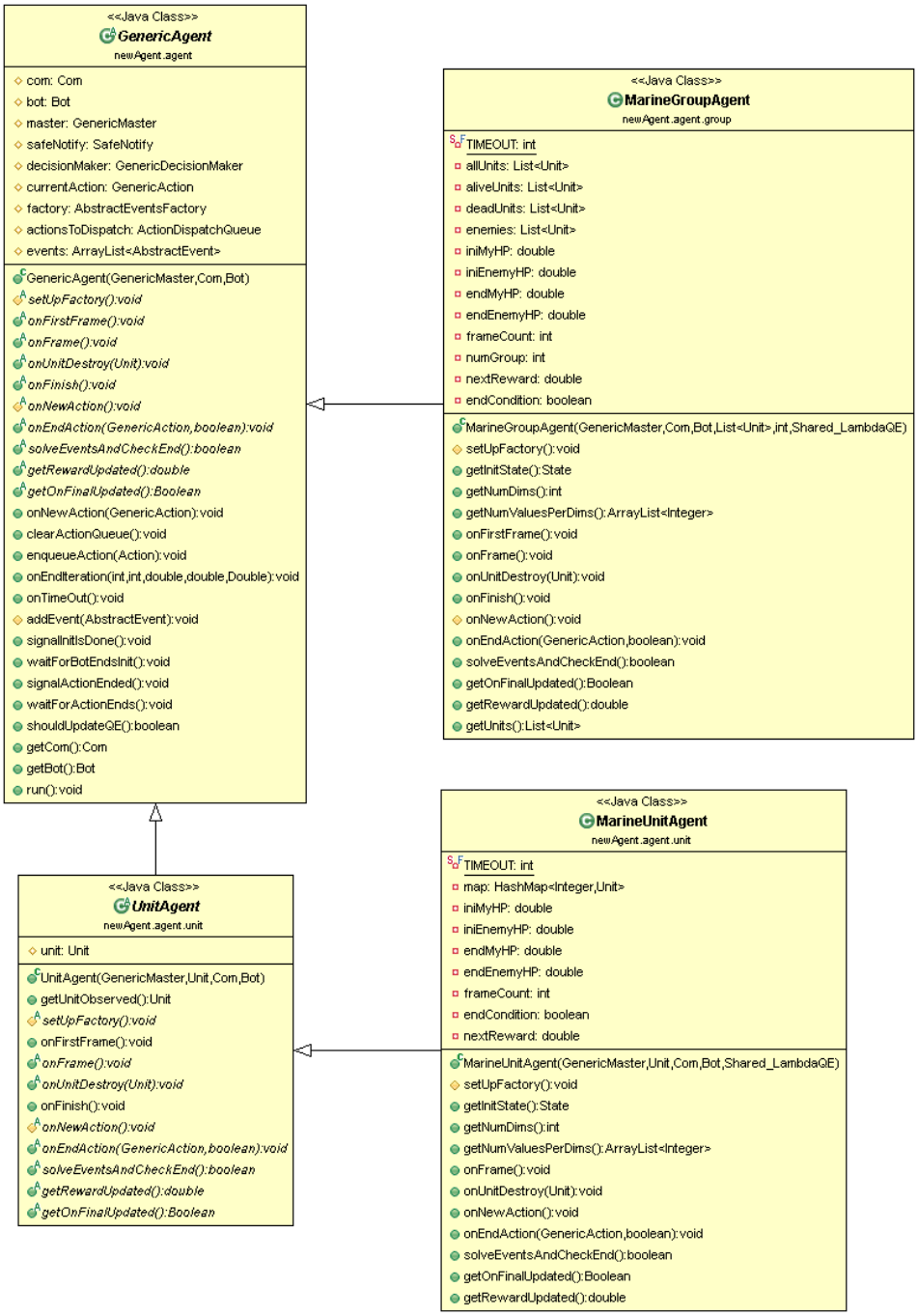


Figura 27 – Diagrama de clases de agentes

4.3.3 Grupos de unidades en varios agentes

Lo último a decir sobre los agentes es, ¿cómo agrupar las unidades en los mismos? Si sólo tenemos un marine que resuelve un laberinto es fácil ya el único agente que tenemos se encarga de ese marine; igual pasa si tenemos un único agente para varias unidades, todas las unidades son controladas por dicho agente. Sin embargo, ¿y si tenemos varias unidades que queremos agrupar para ser controladas por diferentes agentes? Lo más sensato para ello es usar un algoritmo de **clustering**, que en base a las posiciones (x, y) de cada unidad, las agrupe en un número determinado de grupos, y cada uno de esos grupos esté controlado por un agente.

Como el problema es bastante sencillo (en general los grupos de unidades estarán bastante separados y bien definidos) no hace falta un algoritmo de clustering complejo. En concreto, hemos utilizado el *Simple-kmeans* implementado en Weka[17], lo que llevó a un código muy simple de pocas líneas, y que funciona a la perfección.

4.3.4 Acciones

Tenemos dos tipos de acciones: las acciones que utiliza el motor de decisiones, y las acciones que se comunican directamente con BWAPI indicando, por ejemplo, que una unidad debe moverse. Existe una relación directa entre ambas, las acciones del Motor de decisiones se traducen directamente a las acciones del bot.

Por la parte de las **acciones del bot**, el diagrama de clase que las representa aparece en la figura 28.

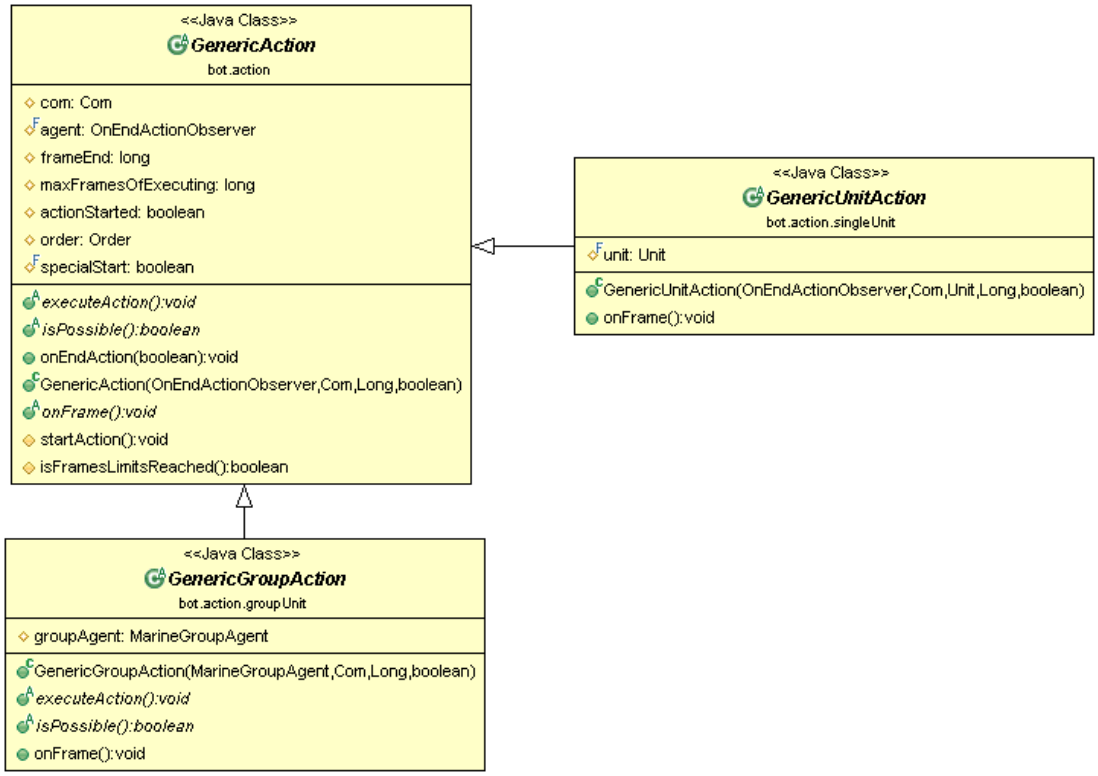


Figura 28 – Diagrama de clases de acciones

Como se puede observar, existe una clase Abstracta *GenericAction,* extendida por las clases *GenericUnitAction* y *GenericGroupAction.*

La primera, como su nombre indica, se utiliza para acciones dirigidas a una única unidad. De esta clase se extienden otras clases que implementan las acciones. En el diagrama de clases de la figura 29 no se muestran todas las acciones, sólo la clase *AttackUnitOnSightLessHP*, que implementa la acción de atacar a la unidad enemiga con menos puntos de vida; y las clases Abstractas *MoveAction,* que permite implementar fácilmente acciones del tipo "moverse a la derecha" o "moverse 20º en sentido horario", y *RelativeMove*, para implementar fácilmente acciones como "acercarse al aliado más cercano" o "alejarse del enemigo"

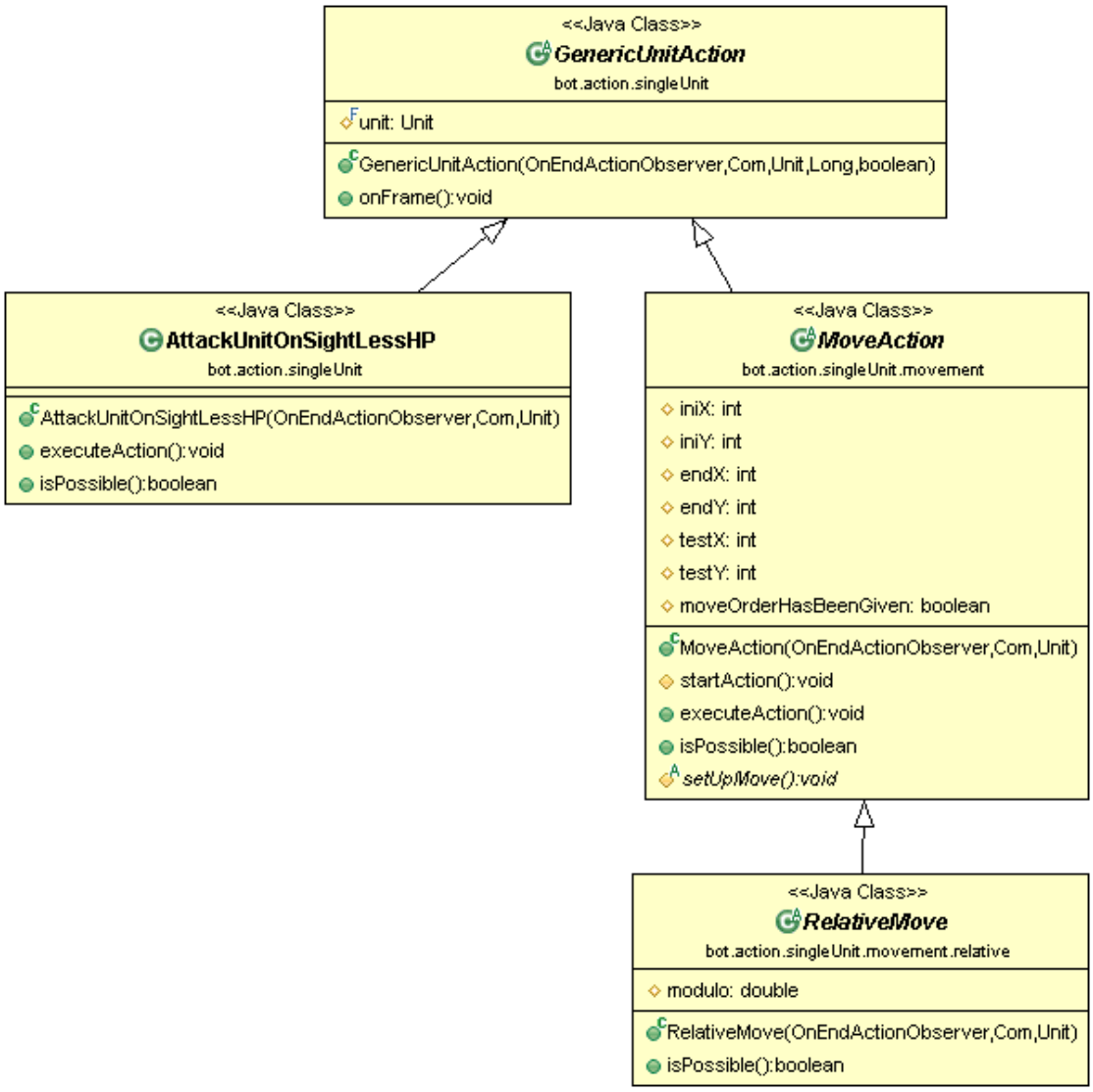
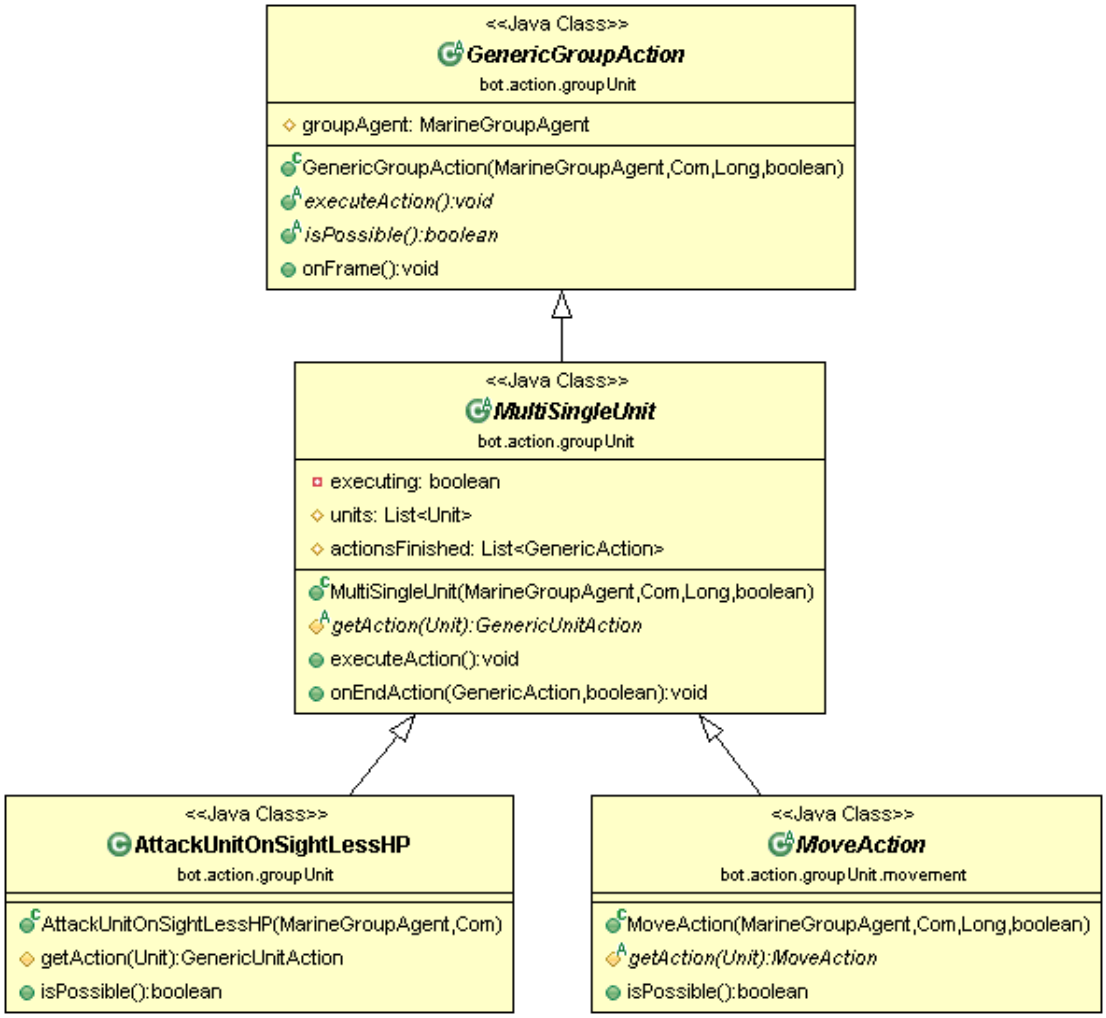


Figura 29 – Diagrama de clases de acciones heredadas

En cuanto a las acciones de grupo, el esquema es similar sólo que aplicadas a un grupo de unidades (controlado por un agente de la clase *GroupAgent*). El diagrama de clases de la figura 30 es similar al anterior. Destaca la aparición de la clase *MultiSingleUnit*, que sirve para implementar fácilmente acciones que ejecuten para cada una de las unidades del *GroupAgent* una de las acciones de *GenericUnitAction*, como por ejemplo hacer que todas las unidades ataquen a la unidad enemiga más cercana. Por supuesto, este esquema se puede extender para implementar acciones más complejas.



Figurar 30 – Diagrama de clases de acciones de grupo

Por otro lado, tenemos las **acciones del motor de decisiones**, que explicamos en la sección del *DecisionMaker.*

4.3.5 Comunicación Agentes - BWAPI

El dilema fue el siguiente, ¿hacemos que toda la lógica se llame desde el Bot, de manera secuencial; o bien mantenemos los agentes (y el super agente) en diferentes Threads, y los comunicamos con señales y memoria compartida?

La primera opción es más fácil de implementar en el aspecto de que no te tienes que preocupar por la sincronización entre los Threads; aunque pensamos que esta implementación daría muchos problemas a la hora de trabajar con varios agentes por lo que finalmente nos decantamos por usar varios Threads. De esta forma, por cada agente tenemos un Thread diferente. Realmente, no está cada Agente (de la clase *GenericAgent*) en un Thread, sino que cada *DecisionMaker* de cada Agente están en Threads diferentes, pero para simplificar el concepto, se puede pensar como un agente por Thread.

Tras probar varias implementaciones más complejas, esta fue la que acabó funcionado mejor (y además es la más sencilla). Simplemente se sincronizan dos aspectos:

- Esperar a que el Bot comience a ejecutar el primer frame. Cada *DecisionMaker* se bloqueará hasta que el Bot pueda enviar al Agente los datos necesarios para empezar a trabajar, generalmente esto es en el primer frame, aunque a veces hay que esperar más.
- Notificar a los agentes cuando una acción ha terminado. Esta señal puede deberse a varias razones como que la acción haya terminado correctamente o que una unidad haya muerto; en general, siempre que el *DecisionMaker* deba de elegir una nueva acción. De esta forma, cuando el *DecisionMaker* elige una acción, queda bloqueado a la espera de esta notificación, y una vez es despertado, toma las recompensas y comprueba si se provoca un estado final o no (todo esto calculado justo antes de mandar la notificación).

4.4 Decision Maker

Por último, tenemos el decision maker. Es la parte encargada de elegir cuál es la siguiente acción a ejecutar. En nuestro caso se realiza utilizando el algoritmo de Q-Learning explicado más arriba.

Como observamos en el diagramas de clases siguiente, Existe una clase abstracta *GenericDecisionMaker* que contiene los métodos utilizados por los agentes, la clase *DM_LambdaQE*, que implementa el Algoritmo Q-Learning(λ) para un sólo agente, y la clase *Shared:LambdaQE*, que extiende al anterior y que se utiliza para varios agentes como veremos más adelante.

Además, tenemos la Intefraz *AbstractQEFunction* y la clase que lo implementa *QEMap*, esta clase es la implementación de la Tabla del algoritmo, tando te la Q como de la E, en

una tabla hash. Funciona independientemente del número de acciones y del estado (tipo y número de direcciones) al usar un direccionamiento genérico:

- **S** es el estado actual
- **A** es la acción actual
- **dimsSize** es el producto de los tamaños de las dimensiones utilizadas para la representación del estado.

```
r = 0;
desp = 1;

for (Dimension dimension : dimensionsInS) {
       r += dimension.discretize() * desp;
       desp *= dimension.getNumOfValues();
}

indiceTablaHash = r + dimsSize * A.ordinal();
```

Básicamente, es el algoritmo de direccionamiento de un array n-dimensional sobre un array unidimensional, con lo que conseguimos que cada par Estado-Acción (quedando el estado determinado por el valor discretizado de sus dimensiones) quede unívocamente definido por un entero.

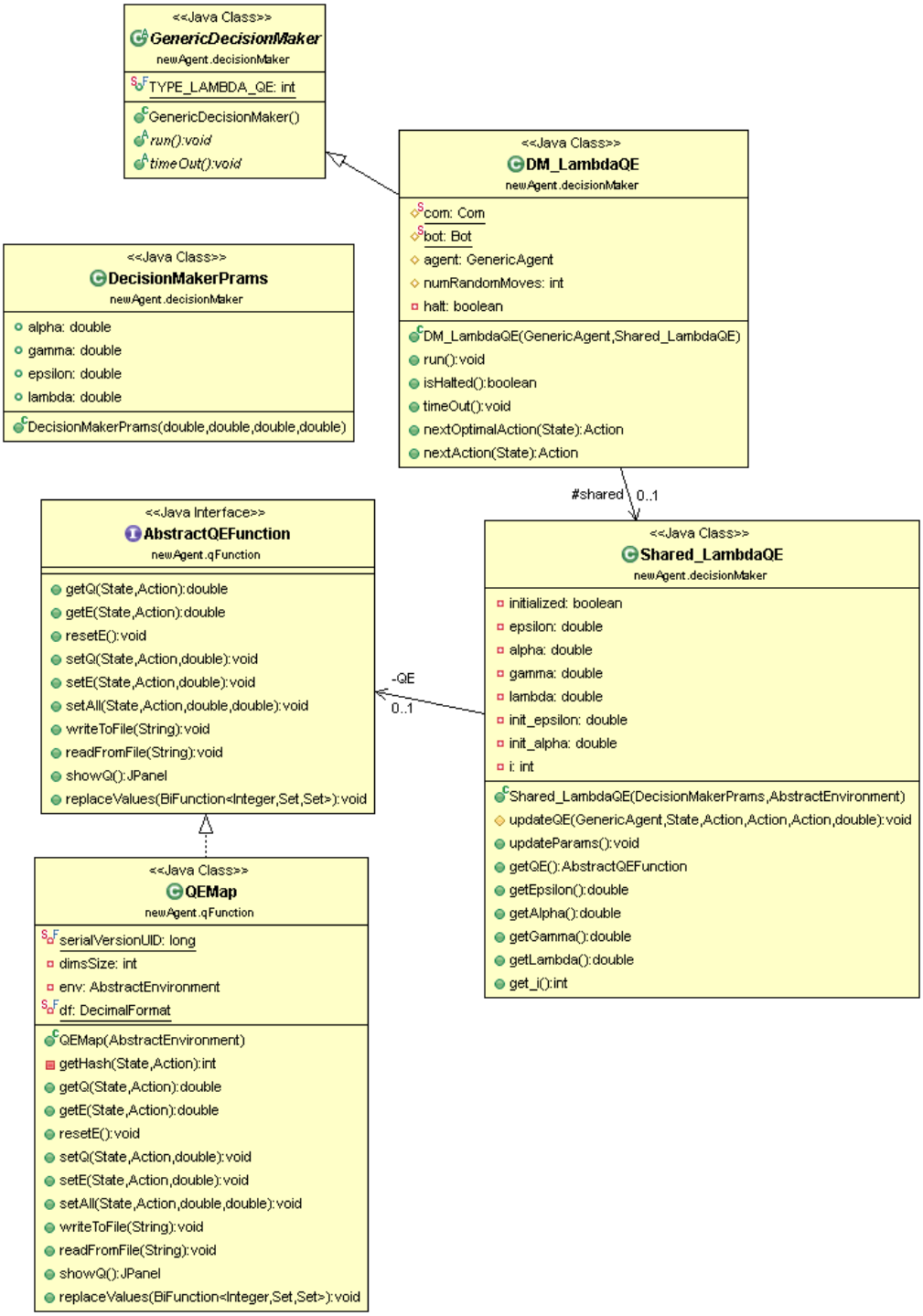


Figura 31 – Diagrama de clases de decissionMaker

4.4.1 Representación de los estados

Una parte muy importante del Q-Learning, y de la Inteligencia artificial en general, es la representación de los estados. Con nuestro Framework crear nuevas representaciones para los estados es una tarea bastante sencilla. Como comentábamos antes, la implementación del algoritmo, así como de la tabla, no necesita ser modificada al cambiar la representación.

Como observamos en la figura 32, la clase *State* agrega una instancia de la clase *StateData*, que es la **representación del estado.**

La representación del estado contiene una **lista de Dimensiones**, cada una destinada a modelar parte de la representación del estado. Estas dimensiones tienen las siguientes características:

- **Un tipo**, que puede ser Integer, Double….
- **Un valor discreto máximo**, siendo el mínimo siempre 0.
- **Un nombre.**
- **Una función discretizadora**, dado un valor del tipo de la dimensión, lo convierte a un entero en el rango [0, max], max incluido (con lo que hay max + 1 valores posibles).
- **Una función *getNewDimension*,** que calcula el valor para esta dimensión, según el estado actual del bot. Utilizado cuando se transita a un nuevo estado.

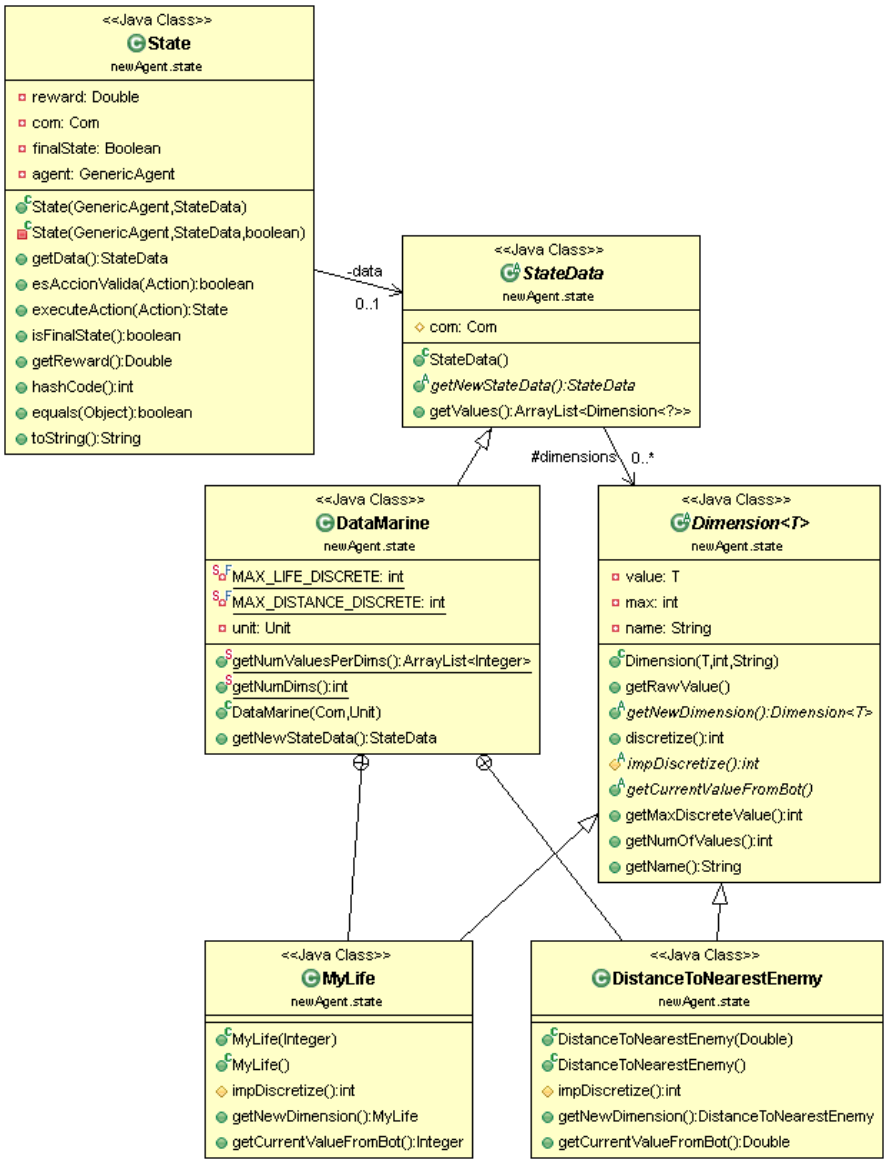


Figura 32 – Diagrama de clases del estado

Implementando todo eso, creamos nuevas dimensiones fácilmente. En el diagrama anterior, como ejemplo, tenemos la clase *DataMarine,* que modela la representación del estado para el agente que controla una sola unidad, y contiene **dos dimensiones:** *MyLife,* que toma los puntos de vida de la unidad actual; y *DistanceToNearestEnemy*, que calcula, como su propio nombre indica, la distancia al enemigo más cercano.

4.4.2 Acciones del DecisionMaker

Previamente hemos hablado de las acciones del bot, ahora hablaremos de las del agente.

Estas acciones son utilizadas directamente por el *DecisionMaker*. Es básicamente un enumerado que contiene un símbolo por cada acción del bot, y que permite traducir de unas a otras. Además, permite seleccionar fácilmente desde la GUI que acciones usar para cada ejecución, sin tener que tocar nada del código.

Esto permite utilizar en el algoritmo de Q-Learning el enumerado fácilmente, iterando por todos los valores deseados, y traducir a una acción del bot cuando sea necesario.

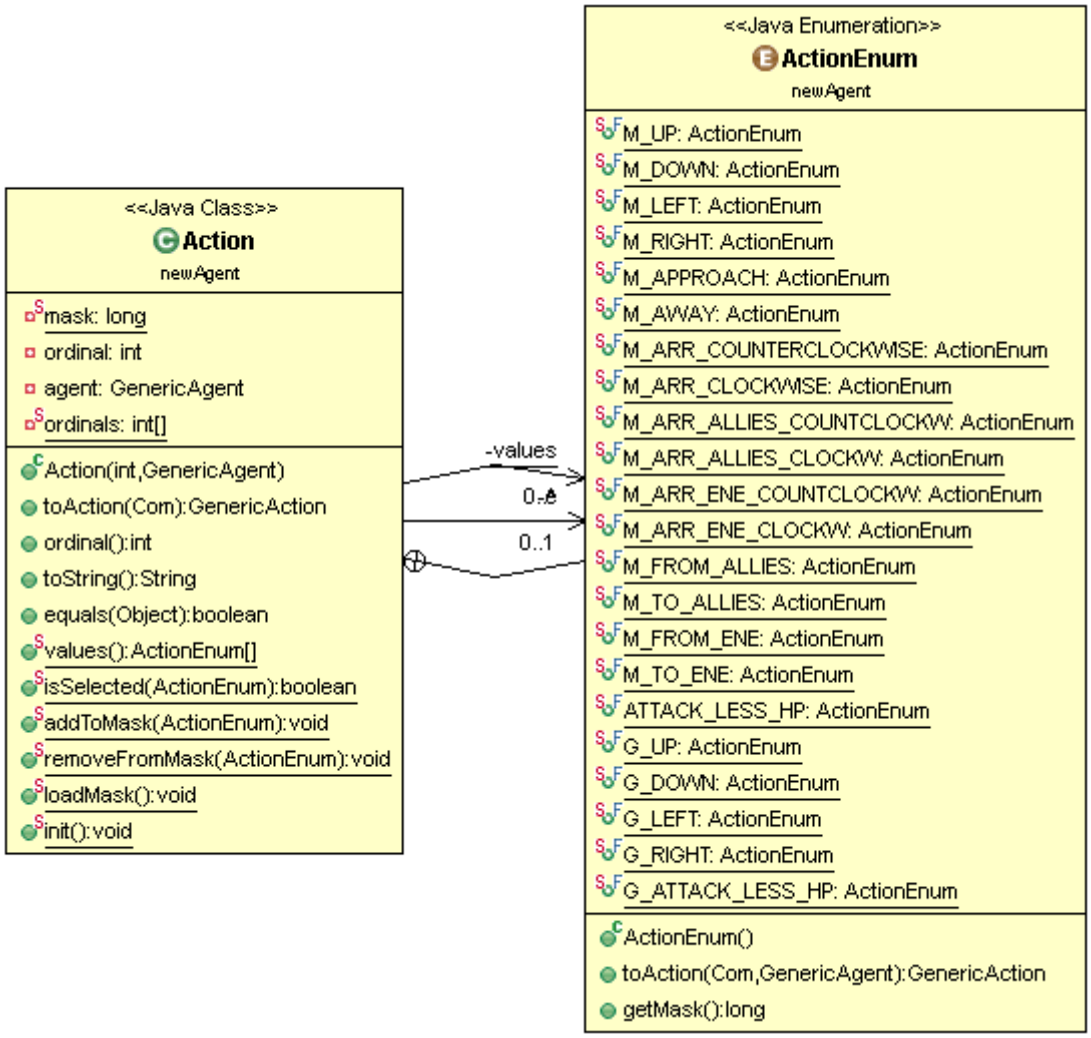


Figura 33 – Enumerado de acciones

4.4.3 Ejecución de Acciones: comunicación con el agente

Un detalle que todavía no hemos comentado, y que está relacionado con los agentes, es cómo se ejecutan las acciones.

Las acciones del *DecisionMaker* son traducidas a acciones del Bot, tal y como se explica en el apartado anterior, y una vez traducidas se **encolan** en la cola de acciones del agente en cuestión. Esta cola será vaciada, y lanzadas a ejecución las acciones de la misma, al inicio del siguiente onFrame del Agente.

4.4.4 Múltiples agentes aprendiendo sobre una misma tabla

Plantear varios agentes aprendiendo sobre una misma tabla plantea algunos problemas:

- Por un lado, carreras de datos, aunque esto se resuelve fácilmente garantizando acceso secuencial.
- Por otro lado, el problema más grave. Si tenemos varios agentes aprendiendo sobre la misma tabla, dando por solucionados los problemas de carreras de datos, se presenta el siguiente problema lógico: si un agente modifica una posición de la tabla, y acto seguido otro la modifica, si esta operación sobrescribe el valor anterior tenemos dos problemas
    - Por un lado, si siempre se realiza en el mismo orden, el agente que va primero nunca refleja su experiencia en la tabla, sólo se refleja la del segundo
    - Por otro, si este orden es aleatorio, nos podemos encontrar con una tabla incongruente: algunas posiciones son la experiencia de un agente, y otras lo son de otro.

Si bien el sistema podría funcionar más o menos bien, lo más sensato es buscar una solución más "justa". En nuestro caso, nos hemos decantado por una muy simple: **alternar el agente que aprende.** De esta manera, en cada iteración del algoritmo, aprenderá un agente, en la siguiente otro, y así sucesivamente se van alternando todos. Así aseguramos que todos los agentes aportan algo de experiencia a la tabla.

Un diagrama de secuencia simplificado de la interacción DecisionMaker – Agente se muestra en la figura 34.

En este diagrama se presenta la secuencia de una interación, desde el lanzamiento de la acción por el *DecisionMaker* hasta su finalización. Un par de comentarios sobre el mismo:

- Entre 9 y 10, en general habrá varios frames, hasta que termine la acción.
- La acción puede terminar por varias razones:
    - Se ejecute correctamente (por ejemplo, llegar a una posición determinada)
    - Se acabe su cuanto de tiempo (un número determinado de frames)

- Se produzca un evento que termine el agente (mueran todas las unidades del agente)

En los dos últimos puntos, es el agente el que finaliza la acción; en el primero, la acción termina por sí sola.

- Al hacer *15: UpdateQTable,* en caso de haber varios agentes, el master tiene que arbitrar la escritura a la tabla
- En caso de llegar a un estado final, se notificaría al master, y el *DecisionMaker* terminaría.
- En caso de no llegar a un estado final, se volvería a iterar como se muestra en el diagrama.

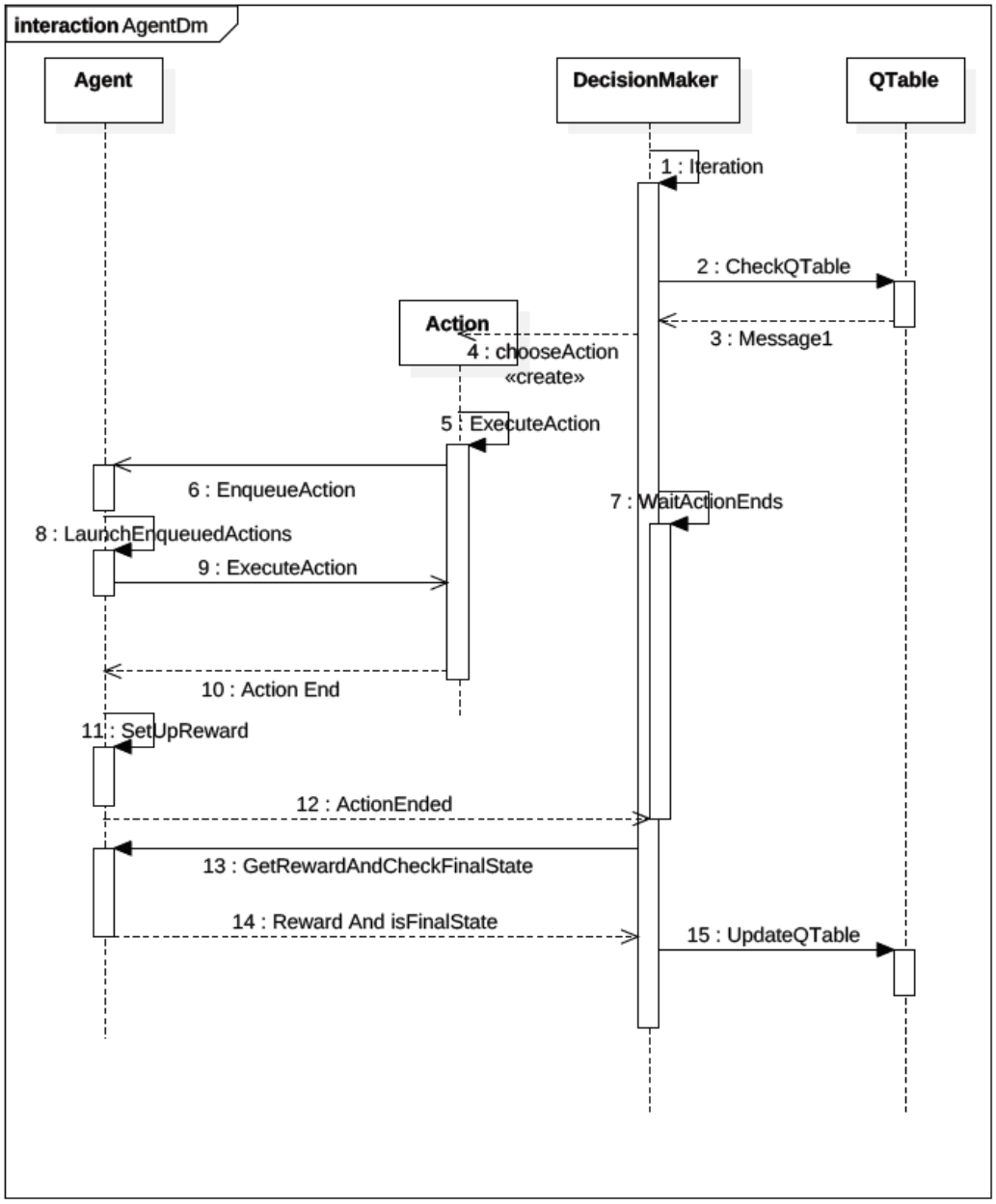


Figura 34 – Diagrama de secuencias de interacciones entre el agente y el decissionMaker

4.5 Utilidades varias

Durante el desarrollo del código hemos realizado algunas partes que no entran en ninguno de los puntos anteriores, pero que merece la pena comentar:

- **Funciones generales** para comprobar el **entorno** de las unidades de BWAPI. Funciones del tipo "unidadesEnemigasCercanas", "hayAliadosAlrededor" o "sumaVidaDeEnemigosCercanos"
- Sistema de **cierre automático de StarCraft** al ejecutar nuestro programa, es de mucha ayuda cuando la vista del juego se queda bloqueada y hay que abrir y cerrar constantemente el programa.
- Lanzamiento automático de *ChaosLauncher* en cada ejecución.
- **Sincronización de los mapas con git.** Con esto hacemos que la carpeta de mapas de StarCraft se mantenga igual que una carpeta en nuestro proyecto sincronizado con git, así podemos compartir mapas fácilmente.
- Sistema de **selección de mapas** en la GUI.
- Sistema de **selección de mensajes de Debug** en la GUI, para elegir qué tipos de mensajes queremos ver (por ejemplo, mostrar las recompensas en cada paso)
- Sistema de **guardado de preferencias** de la GUI, como el último mapa cargado o las acciones que se quieren ejecutar.

4.5.1 GUI

Para facilitar las pruebas, creemos que una GUI es imprescindible, pues trabajar sólo con la consola puede ser insuficiente. De esta manera hemos desarrollado una GUI que, a pesar de ser bastante simple, ofrece una ayuda bastante importante, no sólo por el mero hecho de ver las cosas con mayor claridad, sino por las capacidades que tiene de configurar algunos parámetros de ejecución.

En la vista principal Console [Figura 35], tenemos varias partes:

- Por un lado, los **parámetros del algoritmo de Q-Learning**, que podemos configurar antes de lanzar a ejecución
- Los **valores FPS y EPS**, que nos informan respectivamente de los Frames or segundo en ejecución, y el número medio de ejecuciones por segundo.
- El **parámetro *Speed***, que nos permite controlar la velocidad de ejecución de StarCraft
- El **botón *Run***, que comienza la ejecución
- El **ToggleButton GUI**, para alternar el renderizado o no de la GUI según BWAPI
- El **botón ShutSC**, para forzar el cierre de StarCraft
- Los **contadores**, muertes para el número de muertes de nuestras unidades, Asesinatos para el número de asesinatos, y Rel para la relación entre ambos valores.

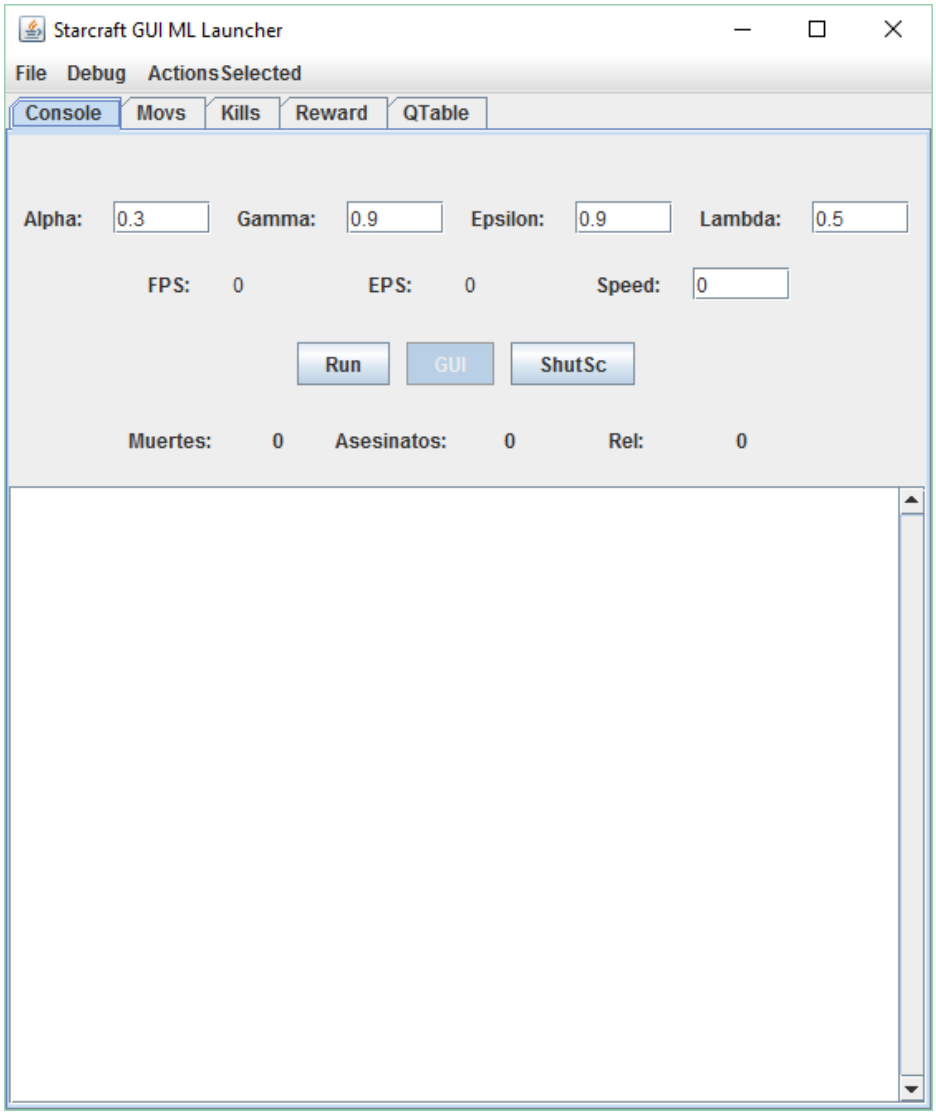


Figura 35 – Vista console de la GUI

- **Menú *File* [Figura 36],** donde podemos elegir el mapa a utilizar, la carpeta de StarCraft y la carpeta de los mapas.

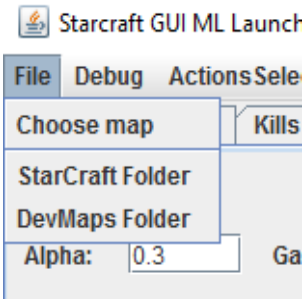


Figura 36 – Menu File de la GUI

- **Menú *Debug* [Figura 37],** donde podemos seleccionar qué mensajes de Debug mostrar en la vista "Consola"

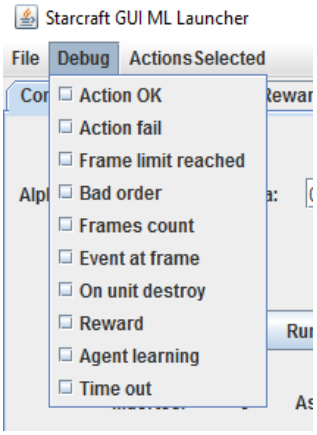


Figura 37 – Menu debug de la GUI

- **Menú *ActionsSelected***, donde podemos elegir qué acciones se podrán tomar en la ejecución.

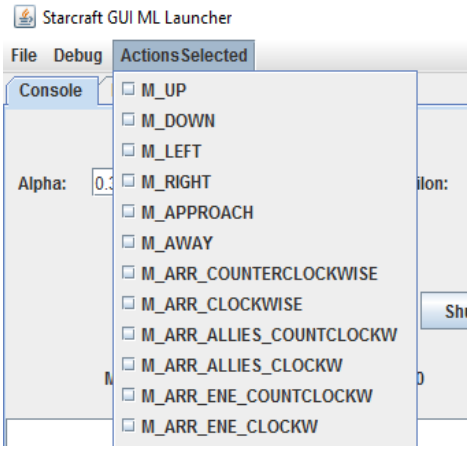


Figura 38 – Menú action selected de la GUI.

El resto de vistas están hechas para proporcionar información adicional del problema concreto que estemos tratando, como por ejemplo gráficas o representaciones de los diferentes elementos del algoritmo. A continuación daremos una pequeña descripción de las vistas:

- **Gráficas - Movs, kills, rewards:** Estas tres pestañas muestran diferentes gráficas que representan características del aprendizaje. La pestaña Movs muestra una gráfica con el número de movimientos por iteración, mientras que rewards y kills muestran gráficas binarias con la densidad de veces que ha ganado el agente. En

el ejemplo siguiente puede verse una gráfica que muestra la densidad de veces que ha ganado el agente.

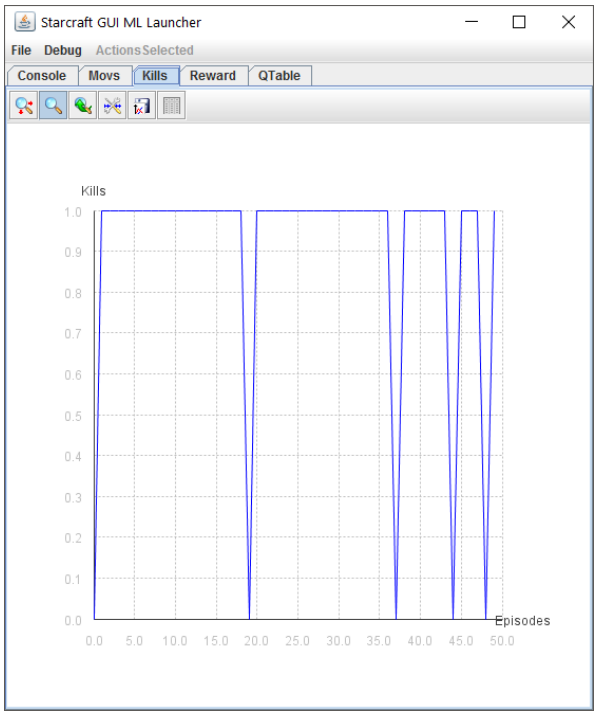


Figura 39 – Vista Kills de la GUI

La idea de estas gráficas es que se puedan adaptar, añadiendo o quitando gráficas, a los diferentes problemas que se vayan a resolver.

- **QTable**: Esta vista nos da una representación de la tabla Q(s, a) del algoritmo, útil para cuando tenemos menos de 3 dimensiones. Entre otras cosas, nos permitió descubrir errores de acceso a la tabla y de actualización, así como ver el aprendizaje de una forma más activa.

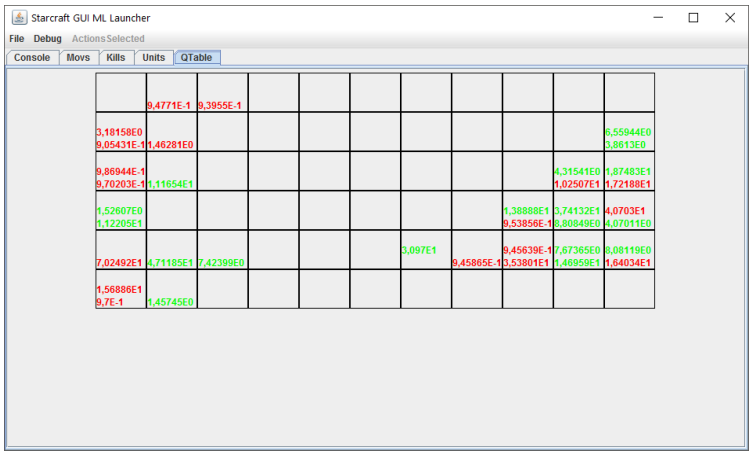


Figura 40 – Vista QTable de la GUI

- **Units**: Esta pestaña nos daba una representación de las unidades alrededor de nuestro agente desde su punto de vista en tiempo real, se actualizaba junto a StarCraft. En rojo aparecen los enemigos y en verde los aliados. En el centro en azul aparece nuestro agente.

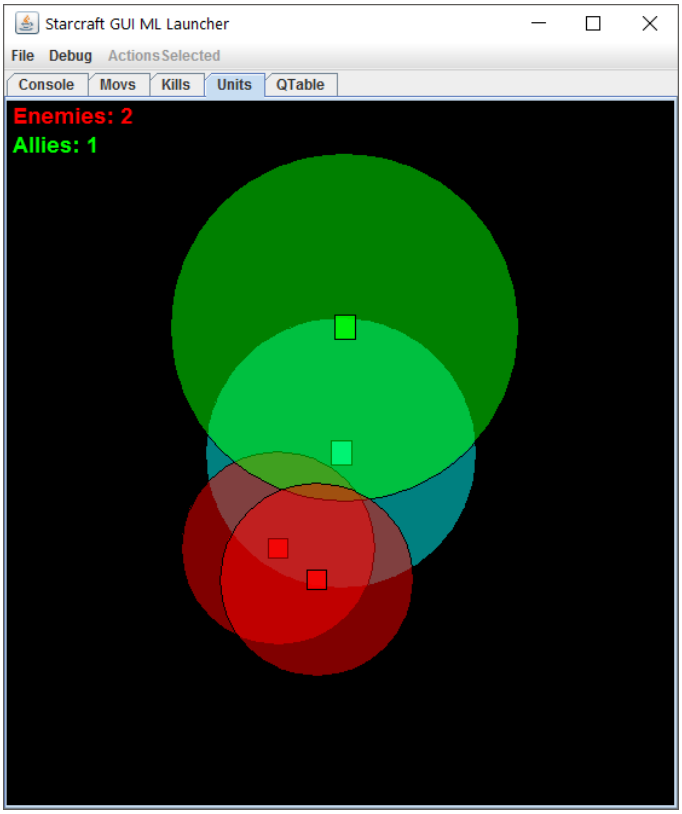


Figura 41 - Vista Units de la GUI

Capítulo 5. Aprendizaje por refuerzo en StarCraft: Brood War

Una vez aprendimos a utilizar el aprendizaje por refuerzo y la BWAPI solo quedaba juntarlo todo.

5.1 Problemas propuestos

5.1.1 Resolución de laberintos

El primer problema que propusimos fue la resolución de un laberinto. La idea para este problema la cogimos del trabajo realizado por otro grupo el año pasado, el cual solucionaba un laberinto con distintas pruebas y trampas. Nuestro laberinto es más sencillo, compuesto solo por paredes y un edificio al que nuestro marine debe llegar.

5.1.1.1 Primer laberinto

Comenzamos realizando un laberinto fuera del entorno de Starcraft [Capítulo 3 – Primer experimento]. El laberinto se componía de un tablero (de dimensiones 3x3 y más tarde 10x10) para probar que nuestro programa alcanzaba la solución óptima y que habíamos implementado el algoritmo Q-Learning correctamente. Con esta prueba superada con éxito solo tuvimos que incluir el algoritmo en el entorno del videojuego.

Así pues, desarrollamos un mapa en el entorno de Starcraft con un marine Terran y un centro de mando que utilizamos como baliza. El marine disponía de un espacio limitado por el que moverse desde el punto de partida (el resto eran paredes), y tenía que encontrar un centro de mando que utilizamos como baliza. Básicamente podía elegir entre cuatro acciones para moverse: moverse arriba, abajo, izquierda y derecha. La acción se escogía mediante un método *greedy,* que buscaba cuál de las cuatro opciones daría mayor recompensa si se siguiese ese camino, basándose en las experiencias anteriores. Además, en el algoritmo existe una probabilidad de no escoger la acción que se debería, sino una aleatoria para explorar caminos nuevos (que podrían ser mejores). Con estas cuatro acciones, el marine tenía que alcanzar la baliza para recibir su recompensa. El juego finalizaba cuando se detectaba una unidad en rango del centro de mando. Repitiendo este proceso varias veces y utilizando lo aprendido en iteraciones anteriores, el marine cada vez realizaba menos acciones para llegar a la baliza.

Las recompensas que obtenía el marine eran de tres tipos: recompensas negativas pequeñas por cada acción que acababa correctamente, recompensas negativas altas por cada acción en la que se quedaba atascado o que no podía terminar (chocarse contra paredes), y una recompensa positiva muy alta si alcanzaba la baliza. Con esto lo que intentábamos era optimizar el recorrido que escogía, de forma que tomase el que requería menor número de movimientos. Al restarle recompensa por cada movimiento, le obligamos a realizar el menor número de movimientos para alcanzar la baliza.

Comenzamos en un mapa pequeño, en el que el territorio disponible para que el marine se moviese era muy limitado. En la figura 42 podemos ver el mapa que utilizamos.

Figura 42 – Primer laberinto en StarCraft

La figura 43 nos muestra el mismo mapa que la imagen anterior pero como se ve en el minimapa. El resto de mapas utilizados no caben en su versión del juego, así que solo se añaden en su versión de minimapa para que se puedan comparar los tamaños. El punto de comienzo es el punto rojo de la esquina superior izquierda en todos los casos, y la baliza es el otro punto rojo.

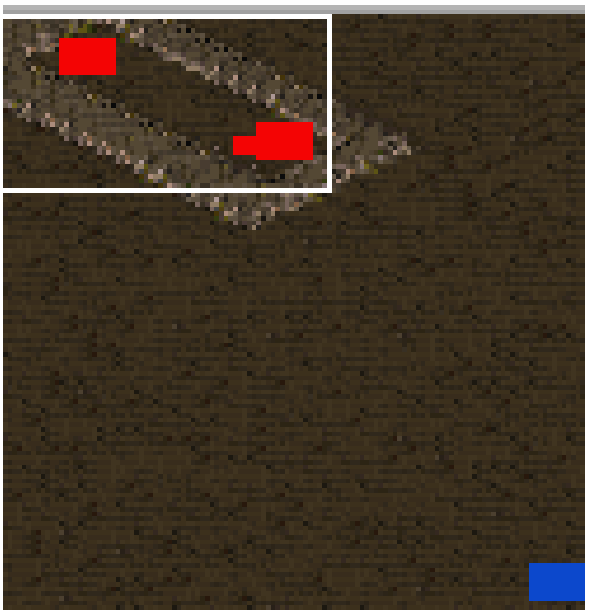


Figura 43 – Primer laberinto (Minimapa)

Al aplicar el algoritmo sobre este mapa obtuvimos los siguientes resultados:

- **Iteración 1**: 84 movimientos en la iteración, de los cuales 7 fueron cogidos de forma aleatoria.
- **Iteración 20**: 40 movimientos de entre los cuales 3 fueron cogidos de forma aleatoria
- **Iteración 50**: 24 movimientos de entre los cuales 2 fueron cogidos de forma aleatoria
- **Iteración 100**: 20 movimientos de entre los cuales ninguno fue cogido de forma aleatoria
- **Iteración 400**: 22 movimientos de entre los cuales 2 fueron cogidos de forma aleatoria

En este primer laberinto, hacer tantas iteraciones no fue útil ya que al cabo de 50 iteraciones, la gráfica se estabiliza entorno a unos valores que varían dependiendo del número de aleatorios. El laberinto se resolvió en un mínimo de 20 movimientos. La figura 44 muestra la relación entre movimientos y aleatorios por el número de iteraciones.

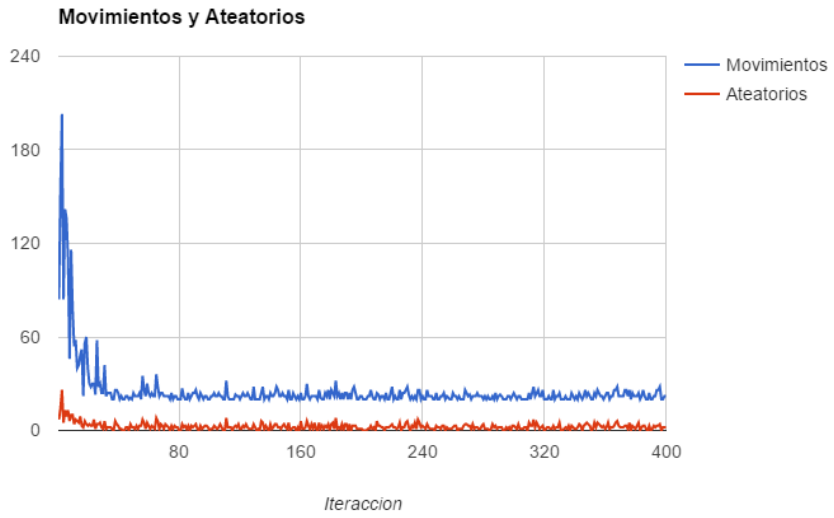


Figura 44 – Gráfica de movimientos

Los picos visibles se deben al porcentaje de acción aleatoria (se intentó explorar un camino nuevo). Se puede ver como el número de aleatorios disminuye a medida que lo hace la cantidad de movimientos.

5.1.1.2 Segundo laberinto

El siguiente mapa que utilizamos fue de mayor tamaño y con dos caminos disponibles desde el punto de salida de los cuales solo uno llevaba a la baliza. Este mapa no era una recta como el anterior, el marine tiene que rodear un muro para poder alcanzar el objetivo. El territorio disponible para explorar ocuparía aproximadamente la mitad del mapa, si lo comparamos con el anterior que ocupaba una cuarta parte:

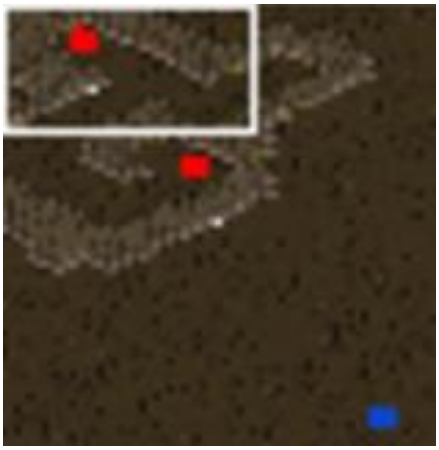


Figura 45 – Segundo laberinto
(Minimapa)

Los datos obtenidos en este mapa del número de movimientos por iteración son estos:

- **Iteración 1**: 526 movimientos en la iteración, de los cuales 52 fueron cogidos de forma aleatoria.
- **Iteración 20**: 179 movimientos en la iteración, de los cuales 17 fueron cogidos de forma aleatoria.
- **Iteración 50**: 67 movimientos en la iteración, de los cuales 8 fueron cogidos de forma aleatoria.
- **Iteración 70**: 47 movimientos en la iteración, de los cuales 6 fueron cogidos de forma aleatoria.
- **Iteración 100**: 43 movimientos en la iteración, de los cuales 2 fueron cogidos de forma aleatoria.
- **Iteración 400**: 49 movimientos en la iteración, de los cuales 8 fueron cogidos de forma aleatoria.

Se observa que en este caso, el número de iteraciones para conseguir una convergencia es mayor que en el mapa anterior. Esta vez se necesitaron alrededor de 70 iteraciones para alcanzar un camino óptimo. Esta vez el laberinto se resolvió en un mínimo de 41 movimientos. La figura 46 podemos ver la relación entre los movimientos y aleatorios por iteración.

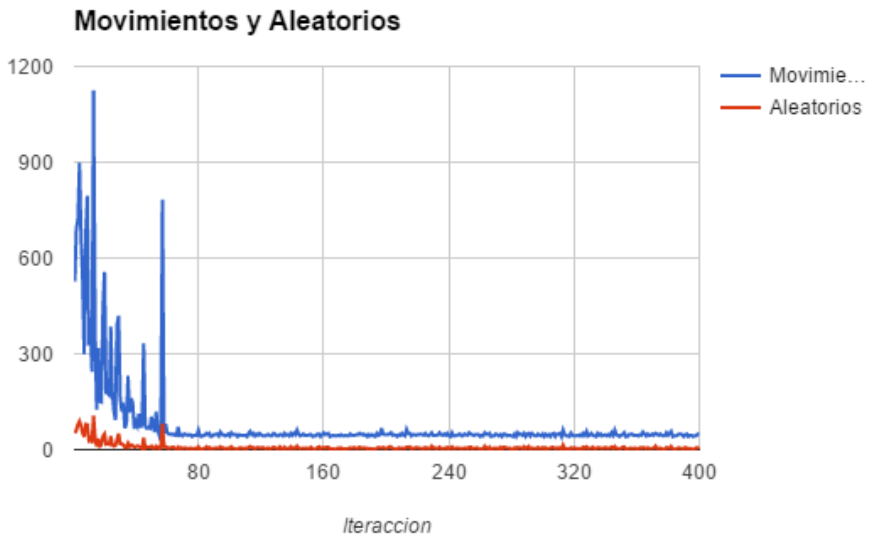


Figura 46 – Gráfica de movimientos

Los picos visibles se deben al porcentaje de acción aleatoria (se intentó explorar un camino nuevo). El pico más significativo de la gráfica se sitúa en la iteración 57 en la que se realizaron 783 movimientos de los cuales 81 fueron aleatoriamente escogidos.

5.1.1.3 Tercer laberinto

El último mapa que desarrollamos fue un laberinto más complejo. Nuestro marine debía escoger entre varios caminos disponibles en un amplio territorio de exploración. En este caso si se equivocaba de camino el número de movimiento que tenía que realizar para volver atrás era mucho más grande que en los anteriores mapas. Se puede observar cómo este mapa se ocupa casi todo el territorio de exploración.

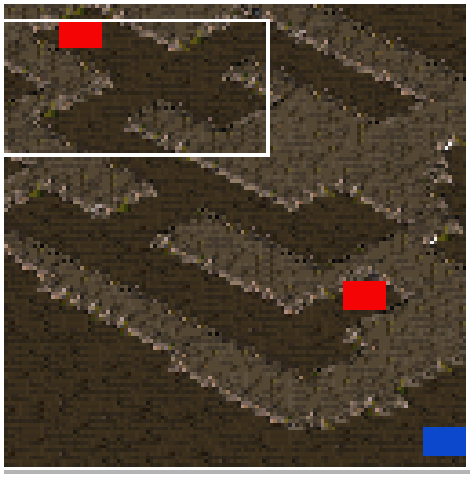


Figura 47 – Tercer laberinto (Minimapa)

En este caso, los datos obtenidos en el proceso de resolución del laberinto fueron más altos que los anteriores debido a que las posibilidades de exploración y movimiento por el mapa eran mucho mayores:

- **Iteración 1**: 4072 movimientos en la iteración, de los cuales 412 fueron cogidos de forma aleatoria.
- **Iteración 20**: 866 movimientos en la iteración, de los cuales 92 fueron cogidos de forma aleatoria.
- **Iteración 50**: 515 movimientos en la iteración, de los cuales 59 fueron cogidos de forma aleatoria.
- **Iteración 100**: 148 movimientos en la iteración, de los cuales 16 fueron cogidos de forma aleatoria.
- **Iteración 150**: 90 movimientos en la iteración, de los cuales 11 fueron cogidos de forma aleatoria.
- **Iteración 400**: 88 movimientos en la iteración, de los cuales 11 fueron cogidos de forma aleatoria.

En este último mapa, el número de iteraciones necesarias para estabilizar la gráfica era mucho mayor que en los mapas anteriores. La gráfica de la figura 48 converge al llegar a 110 iteraciones aproximadamente. También subía mucho el tiempo en ejecución, ya que el número de posibilidades explorables era muchísimo más grande que en los casos anteriores.

El mínimo número de movimientos registrados en este mapa fueron 70 y la gráfica resultante fue la mostrada en la figura 48.

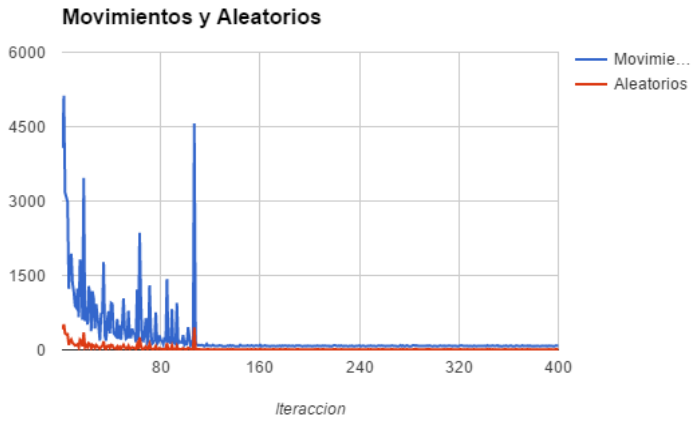


Figura 48 – Gráfica de movimientos

Como se puede ver en la gráfica de la figura 48, el pico más significativo en cuanto a interés se produce en la iteración 108, donde se registraron 4563 movimientos, de los cuales 454 fueron aleatorios.

Una vez terminamos las pruebas en el laberinto, pasamos a introducir un nuevo objetivo: destruir un objetivo inmóvil.

5.1.2 Destrucción de objetivos inmóviles

El siguiente objetivo que cumplimos fue el de destruir una estructura, de forma que el objetivo no pudiese perseguir ni huir de nuestra unidad.

5.1.2.1 Estructura en laberinto

Al principio no queríamos desviarnos mucho de la idea de ampliar el laberinto e incluir nuevas pruebas dentro del mismo. Estas pruebas se resolverían antes de alcanzar la baliza, de forma que la baliza solo estuviese disponible al cumplir todos los requisitos. La primera prueba en la que pensamos fue en añadir una estructura inmóvil en un punto del mapa para que nuestro marine la destruyera. Después de esto, tendría que alcanzar la baliza para finalizar la misión y recibir su recompensa.

Con este objetivo, añadimos una acción más, la de atacar a un objetivo visible. Esta acción solo se podía llevar a cabo si se encontraba algún enemigo en el rango de visión, y se realizaba un ataque corto de forma que el aprendizaje resultase más costoso. Gracias a esto, nos dimos cuenta que nuestra IA no aprendía bien debido a las recompensas negativas que se le daban. Este problema se resolvió quitando estas recompensas y dejando únicamente la recompensa final cuando se alcanzaba el objetivo.

Inicialmente, se creó un mapa como el de la figura 49, que básicamente se componía del segundo laberinto desarrollado para el objetivo anterior, pero con una torreta enemiga que nos atacaba. Esta torreta se encontraba en el camino que llevaba hasta la baliza, con la intención de que la destruyese de camino al objetivo.

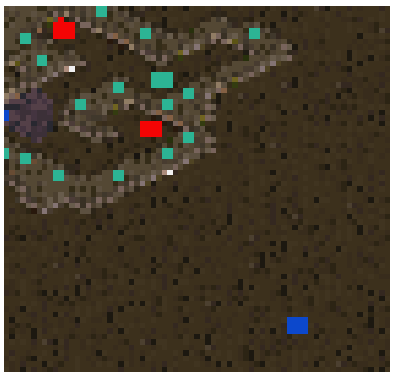


Figura 49 – Destrucción de objetos inmóviles (Minimapa)

En la figura 49, el cuadrado rojo de arriba es el punto de partida del laberinto, el cuadrado rojo de abajo es la baliza a la que debía llegar, y el punto azul de la izquierda del borde (dentro del laberinto) es la torreta enemiga. Lo puntos azules claros son estructuras que no aportaban nada a la resolución del laberinto, que solo nos permitían a nosotros tener visión en todo el laberinto todo el rato.

Al probar este mapa, nuestro marine obviaba la torreta e intentaba ir directamente a completar el laberinto. Tras cambiar las recompensas que se le daba, seguía sin funcionar correctamente. Decidimos depurarlo implementando un nuevo mapa en el que solo debiese destruir una estructura enemiga.

5.1.2.2 Destrucción de estructura estática

En este mapa únicamente se encontraban nuestro marine y una estructura enemiga, encerrados en un espacio pequeño para que las posibilidades de movimiento fuesen pequeñas. Se puede observar nuestra unidad de color rojo y, la torreta de color azul y la orden que se está llevando a cabo (atacar). El mapa apenas era el doble de lo mostrado en la figura 50.

Figura 50 – Marine atacando estructura (Colonia de esporas)

Esta estructura no atacaba, y por tanto tarde o temprano acabaría siendo destruida por nuestro marine. Era una prueba muy simple que finalizaba cuando se detectaba que una unidad o estructura había sido destruida.

Se puede observar en la figura 51 cómo, al igual que en el laberinto, al aumentar el número de interacciones la cantidad de movimientos que realiza la unidad es menor.

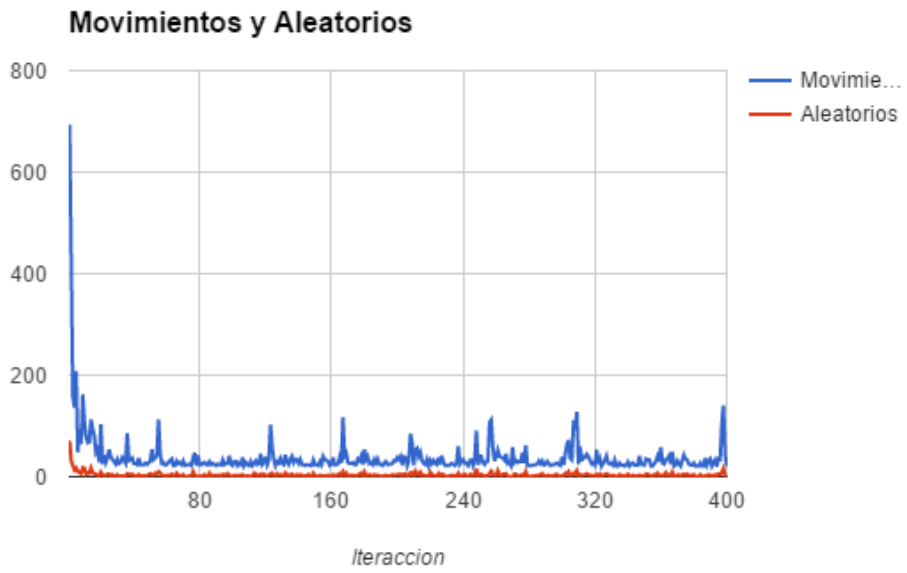


Figura 51 – Gráfica de movimientos

Las iteraciones se estabilizan alrededor de la iteración 25 y alcanza un mínimo de 20 iteraciones con 0 aleatorios.

- **Iteración 1**: 694 movimientos en la iteración, de los cuales 71 fueron cogidos de forma aleatoria.
- **Iteración 5**: 136 movimientos en la iteración, de los cuales 13 fueron cogidos de forma aleatoria.
- **Iteración 25**: 37 movimientos en la iteración, de los cuales 5 fueron cogidos de forma aleatoria.
- **Iteración 100**: 22 movimientos en la iteración, de los cuales 1 fue cogido de forma aleatoria.
- **Iteración 150**: 33 movimientos en la iteración, de los cuales 7 fueron cogidos de forma aleatoria.
- **Iteración 400**: 21 movimientos en la iteración, de los cuales 1 fue cogido de forma aleatoria.

5.1.2.3 Destrucción de estructura hostil

Para complicar el problema, sustituimos la estructura enemiga por una torreta que atacaba a nuestro marine siempre que éste entraba en rango. Esta torreta disponía de más daño y aguante que nuestro marine, así que siempre lo mataba. Para igualar la situación, disminuimos mucho el daño de la torreta, ya que mientras que ella atacaba a nuestro marine siempre, este se movía y atacaba de forma alternada.

Figura 52 – Marine atacando estructura (Colonia hundida)

En la imagen de la figura 52 podemos ver al marine moviéndose mientras está siendo atacado por la torreta. El daño de la torreta fue disminuido a 1 de daño por ataque, mientras que el daño del marine se mantuvo.

En este mapa, el juego seguía finalizando cuando una unidad moría, ya fuese la torreta o nuestro marine. El número de movimientos que se realizasen ya no era tan importante ya que el objetivo que perseguíamos en esta prueba era que nuestra unidad sobreviviese. Aún así, el número de movimientos disminuye con cada iteración, debido a que la forma de destruir la torreta era atacándola continuamente. En caso de que no fuese así, el marine moría debido a que se quedaba dando vueltas mientras que la torre le atacaba, o si se salía de su rango de ataque, la torreta recuperaba vida mientras que el marine no lo hacía.

Con respecto a la relación entre muertes y asesinatos, cabe destacar que en las primeras iteraciones nuestro marine moría en casi todas las ocasiones hasta que aprendía a atacar de continuo. Una vez lograba destruir la torreta un par de veces, el aprendizaje se volvía mucho más rápido y conseguía ganar en casi todos los episodios restantes.

Después de estas pruebas, decidimos que sería más interesante sustituir las estructuras enemigas por unidades móviles, de forma que las posibilidades de perder fuesen más altas que en una resolución de un laberinto. De esta forma, aportaríamos también datos nuevos y distintos de los que se presentaron en el proyecto del año anterior.

5.1.3 Destrucción de objetivos móviles

Una de las primeras situaciones que ideamos para este problema, fue utilizar como contrincante un marine con exactamente las mismas características que el nuestro, sin embargo, nuestro marine conseguía ganar siempre casi desde el principio, por lo que intentamos pensar otra alternativa.

Pensamos también en poner diferentes tipos de unidades, pero dado que la velocidad de las unidades que atacan suele ser superior o igual a la del marine, el marine no podía aprender estrategias del tipo alejarse y atacar cuando el contrincante estuviera cerca. Las unidades siempre eran o más débiles, por lo que aprendía a atacar continuamente hasta matarle, o más fuertes, por lo que aprendía a atacar hasta morir.

Todo esto nos llevó a idear algún mecanismo más elaborado al que el marine pudiera recurrir cuando necesitara recuperar vida, expuesto en el siguiente apartado.

5.1.4 Sistema curar-atacar

En este problema comprobaremos que el agente sea capaz de aprender cuando necesita curarse, antes de seguir atacando. Como ya dijimos en anteriores capítulos, si enfrentamos en una batalla a un Zergling con un marine, el vencedor será el marine.

En este mapa, tendremos un marine, en frente suya dos Zergling, y a sus espaldas a un médico. El objetivo del marine será derrotar a ambos Zerglings con la ayuda del médico. Cabe destacar el hecho de que el médico tiene energía, la cual gasta a la hora de curar, por lo que el marine deberá actuar rápido para derrotar a sus enemigos antes de ser derrotado.

Figura 53 – Problema curar-atacar

A la hora de resolver este problema, nos encontramos con el inconveniente de que el algoritmo de Q-learning que teníamos implementado era el algoritmo Q-learning de un solo paso, por lo que en lugar de aprender secuencias de acciones que lleven al agente desde un estado inicial hasta uno final, solo aprendía acciones que llevaban a estados finales, propagando lentamente en cada iteración paso a paso lo aprendido hacia atrás. Para conseguir solucionar este problema, empezamos a utilizar el algoritmo Q-learning (λ), que permite aprender directamente secuencias de acciones.

El conocimiento que tendrá el marine de su entorno a la hora de tomar decisiones consistirá en la distancia al enemigo más cercano, y su propia salud en cantidad de HP. Para simplificar el problema, hemos hecho que ambos parámetros están discretizados en una escala del 1 al 10, siendo 1 que el marine tiene poca salud o que el Zergling está muy cerca, y 10 que el marine tiene mucha salud y el Zergling está muy lejos.

Las acciones que tendremos en este problema serán las siguientes:

- Atacar enemigo con menos salud.
- Acercarse a aliados.
- Alejarse de enemigos.

Dando un número de estados posibles para el agente de 10*10*3 = 300.

La estrategia que aprende el agente es directamente retroceder hasta ponerse al lado del médico para que le cure mientras mata a los dos Zergling, consiguiendo así matar a ambos rápidamente. Lo interesante de esto no es el hecho de que consiga matar a ambos Zergling, sino que la mayoría de las veces aprende estrategias que consiguen explotar las características del juego de manera que uno de los dos Zergling se queda bloqueado sin atacar mientras el otro le ataca, lo que desemboca en una pérdida de vida más lenta por parte del marine al ser sólo atacado por un enemigo, provocando que el médico tenga más fácil curarle.

En la gráfica de la figura 54 se puede ver que el agente aprende de una manera bastante rápida, tanto que tras probar 100 veces, aproximadamente entre las iteraciones 7 y la 13 ha conseguido una solución ganadora y empieza a repetirla a partir de entonces, tomando eventualmente algún movimiento aleatorio que le lleva a perder, problema que se soluciona al utilizar la función de temperatura:

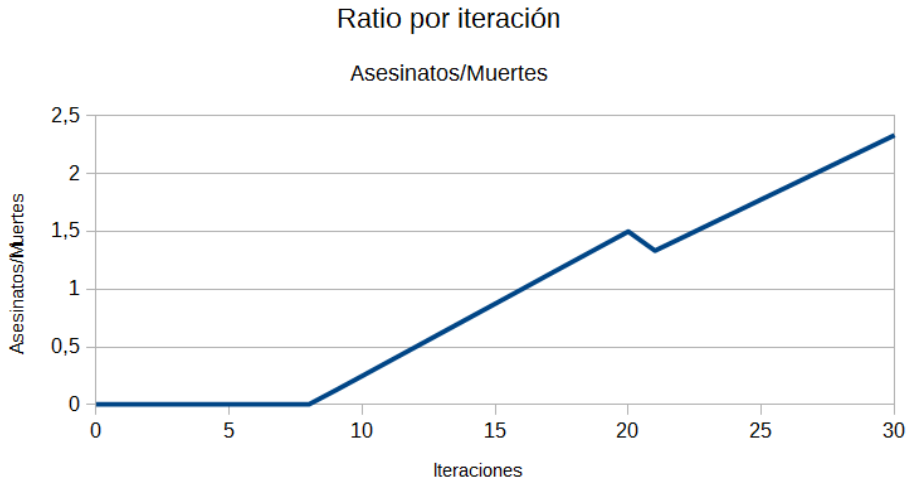


Figura 54 – Gráfica asesinatos/muertes

En la figura 54 puede observarse también que aproximadamente a partir del doble de iteraciones de la iteración en la que consigue ganar, empieza a ganar más de lo que pierde.

6. Conclusions and further work

6.1 Conclusions

As we said in chapter 4, one of the main challenges of this project has been the coding and use of BWAPI, in which we have unfortunately expended much more time than making experiments with reinforcement learning. That's why we will discuss in this section the conclusions and further work about reinforcement learning and framework as well.

Our conclusions about our framework are as follows:

- Working with BWAPI directly is a tedious task. As we have been developing our framework, it became easier to make experiments with reinforcement learning (once errors were fixed).
- Although we had our framework and we tried to do our best, it's necessary to keep on working with it, to reach a maintainable and full-working code, allowing it to be extended.
- Generally, as more abstract the layer over we work is, longer is the time we can dedicate to work with reinforcement learning.
- One of the easiest ways to simplify the use of our framework is through the GUI; as more we control the features of BWAPI through the GUI, easier is the work.

Conclusions about reinforcement learning:

- This field of machine learning is extremely complex. Even the easiest problem, the maze, was problematic. For example, the abstract idea we had for solving the maze (Chapter 3 - First experiment) was to walk over the cells until the final state was reached. In Starcraft, these concepts do not exist, basically, the unit moves around pixel by pixel (or block by block of N pixels).
- In relation to problems involving enemies, the greatest problem was the state representation. If we think in a human being playing StarCraft, when we are in a situation of combat, we analyze too many variables over our units and the enemy units to know if we can beat the enemy in the most optimal way, observing which types of units there are, how they are disposed, how many of them there are… This leads us to a table size too difficult to manage, it is a challenge to code it, for example, how can we analyze the position of the units to define a discretizable state?
- The Q-Learning(0) algorithm it's not enough for experiments rather than the maze one. In complex problems, like the soldier-medic-zerg one, the sequence of actions it's very important, so the TD(λ) it's much more suitable.

- When applying the Q-Learning algorithm on other domains, it needs to be changed a little bit. For example, in the Tic-Tac-Toe problem, the algorithms must wait until the opponent finishes his movement, therefore threads synchronization is essential, and due to the amount of possible states, getting rid of the impossible ones is essential to improve performance.

6.2 Further work:

The part of our project which needs more improvement is the reinforcement learning one, although the framework is

About the framework:

- Improve some parts of the code. More specifically:
  - The translation DecisionMaker-Action to Bot-Action. Now it's using the *instanceof* operator, maybe i'll be better to use a Factory pattern, where the factory is provided by the agent.
  - Improve the GUI, for example implement one Window per each agent could be very useful. Also, improve the console tab, saving the content onto a file, and improve the scroll.
  - Permit launching a new StarCraft instance without restarting the GUI.
- Extend the functions that can be modified within the GUI, for example having a scroll to vary the in Game speed, from 0 (maximum speed) to 60 (a fairly slow speed), without having to write the number, as is implemented now.
- Change some GUI components to match what they are representing, for example number input should be spinners.
- Fix some errors:
  - Sometimes, the first execution frame, where some calls are made to BWAPI, the API raises an exception like "invalid memory address". Our guess here is that, for any reason, on that frame not all information is present, or is not fully valid. We have tried to force the program to wait several frames, making game pauses, but nothing seemed to work well. Here are some situations where the error occurs:
    - When an attack action pretends to be executed, it looks for enemy units in range, which it ends on calling *isAlly* function of BWAPI. This calls causes the error mentioned above.
    - When working with units groups, as the information seems to be incomplete as we already said, it's impossible to create the desired number of groups.
- We weren't able to make the group agent to work correctly: this is partially provoked by the above errors, we have not been capable to experiment with unit's groups, and nothing more than keep different groups executing move actions independently.

- To create more actions, for example for groups create something like “attack every unit to the most powerful enemy unit” or “change the ally unit which is taking damage, in order to prevent it from dying”.
- Improve the system for allow easy agents hierarchy.

About the reinforcement learning:

- Research in new machine learning methods, for example Evolutionary algorithms or match analysis, and perform some comparison between results.
- In the temperature functions, make that the potential value depends on the obtained rewards, i.e. the potential value depends on the distance of the distance to an optimal-optimistic value of the last N obtained rewards, making epsilon increase faster, and alpha decrease faster.
- Improve general performance. Specially, the Q-Table saves *all* the states, which is pretty naive, as some of that states may be never visited; and the E table is pretty bad optimized too. In these regards, there are several actions to take: to group states if they are very similar (which leads to the question: when is a state similar to another?), add the states to the table only where they are first visited, etc.
- Research in new state representations, e.g. consider enemies position, not only their distance to the ally unit. This might be implemented by a “Danger matrix”, for example a 3 x 3 where each position evaluates the danger of that position on the map.
- About group coordination, or specifically agent’s coordination, the simplest idea is a kind of an agent hierarchy, for example an “Army” agent that takes decisions like “unit’s groups one and two, attack the enemy frontline, while unit 3, which have long range units, attacks far away enemy units, this action would imply that units 1 and 2 moves forward and 3 behind, and each of these agents will perform those actions as they have learned by itselves.

As final thoughts, although this domain is really complex, reinforcement learning is a great and powerful machine learning method. From a computational point of view, we think that it's a good solution to the machine learning problem, that solves some problems that supervised and unsupervised machine learning suffers, for example the fact that reinforcement learning doesn't need a big data set to be trained appropriately. We don't mean that reinforcement learning is a substitute, we rather say that it's a great complement for other machine learning approaches.

The three of us we all agree that it will be very interesting to have a subject based on reinforcement learning, as almost everything of what we know about it we have to find all information by ourselves. While it's true that on *Artificial Intelligence* subject we do

study machine learning, it only introduces briefly reinforcement learning, a theoretical introduction that doesn't deepens in the algorithms, and such introduction it's not enough for a work like this.

About the use of reinforcement learning on StarCraft, and video games in general, we think that is a field to research a lot, where a lot of useful and interesting results can be found, that could lead to games that learns from the player and adapts to him, for example varying the game difficulty regarding the player actions, and improving overall game satisfaction.

And now, we would like to remind the reader that all our framework code is available on GitHub with a CopyLeft license, and therefore could be improved, and used in future works.

https://github.com/TFG-StarCraft/TFG-StarCraft

We thank the reader interest and dedication when reading our paper, and we wish him well.

Capítulo 6. Conclusiones y trabajo futuro

6.1 Conclusiones

Como señalamos en el apartado 4, uno de los mayores desafíos de este proyecto ha sido la programación y el uso de BWAPI, que nos ha quitado mucho tiempo de hacer pruebas con el aprendizaje por refuerzo. De esta manera, en esta sección no hablaremos sólo de las conclusiones y el trabajo futuro sobre el Aprendizaje por refuerzo, sino también del *framework* que hemos desarrollado.

Por la parte del framework, las conclusiones son las siguientes:

- Trabajar directamente sobre BWAPI es realmente tedioso. A medida que hemos ido desarrollando nuestro framework, se nos hacía más fácil realizar pruebas con Q-Learning (una vez arreglados los fallos).
- A pesar de contar con nuestro *framework* y de intentar hacerlo lo mejor posible, hay que seguir dedicando tiempo a trabajar sobre el mismo, para que el código sea mantenible y funcione, permitiendo que se siga extendiendo.
- En general, cuanto más abstracta sea la capa sobre la que trabajamos, más tiempo podemos dedicar a trabajar sobre el Aprendizaje por Refuerzo.
- Una de las formas más sencillas de simplificar el uso del framework es mediante la GUI, a medida que controlábamos más aspectos de BWAPI y del aprendizaje por refuerzo desde la GUI más se simplifica el trabajo.

En la parte del Aprendizaje por Refuerzo:

- El domino sobre el que estamos trabajando es extremadamente complejo. Incluso el laberinto, el problema con diferencia más sencillo, al principio dio problemas, por ejemplo, la idea abstracta que tenemos para resolver un laberinto [Capítulo 3 - primer laberinto] es ir recorriendo casillas hasta llegar al final. En la parte del laberinto en *StarCraft* todos estos conceptos no existen, básicamente la unidad se movía prácticamente de píxel en píxel (o en bloques de N píxeles).
- En cuanto a los problemas que involucran a enemigos, un gran problema es la representación del estado: si pensamos a uno de nosotros mismos jugando al juego, cuando nos encontramos en una situación de combate, analizamos muchas variables para saber, no sólo si podemos vencer al contrincante, sino cómo y cuál es la manera más óptima: ver qué tipos de unidades hay, cómo están colocadas, cuántas hay… esto aplicado tanto a las unidades enemigas como a las propias. Esto no lleva sólo a un tamaño de la tabla descomunal / imposible de manejar, sino también a un desafío para programarlo, por

ejemplo ¿cómo analizar las posiciones de las unidades para poder definir un estado discretizable?

- El algoritmo Q-Learning(0) no es para nada suficiente cuando nos salimos del problema del laberinto. Al fin de al cabo, en los problemas más complejos influye mucho la secuencia de acciones que se ha llevado a cabo, por lo que el uso del algoritmo Q-Learning(λ) es mucho más apropiado.
- A la hora de aplicar el algoritmo de Q-Learning en otros dominios hay que modificar ligeramente el algoritmo. Por ejemplo, en el 3 en raya, el algoritmo tiene que quedar bloqueado hasta que el jugador contrincante haya realizado un movimiento, en este caso la coordinación entre hilos es crucial, y dado a la gran cantidad de estados que puede haber, delimitar los no posibles para reducir su número puede aumentar muchísimo el rendimiento.

6.2 Trabajo futuro

Lo que más queda para ampliar es el propio estudio del aprendizaje por refuerzo, aunque también se puede mejorar el Framework.

Respecto a la parte del framework:

- Mejorar algunas partes del código. En concreto:
  - La traducción Acción del *DecisionMaker* - Acción del bot. Quizás sea mejor utilizar una factoría implementada por el agente que se está utilizando, en lugar de usar *instanceof*, aunque esto seguramente dificulte algo la conexión con la GUI.
  - Mejorar la GUI, por ejemplo una ventana por agente podría ser de gran ayuda. Además de esto, mejorar la pestaña de la consola, guardar el contenido en un fichero de texto y mejorar el *scroll*.
  - Permitir lanzar una nueva ejecución de StarCraft sin tener que reiniciar la GUI.
- Extender las funciones que se pueden modificar desde la GUI, por ejemplo, un scroll que permite variar la velocidad de 0 (máxima) a 60 (normal), sin tener que meter un número a mano.
- Adecuar los componentes de la GUI a los que mejor se adapten, por ejemplo, las entradas de números deberían ser spinners.
- Reparar algunos errores:
  - A veces, en el primer frame en el que se realizan llamadas a BWAPI, ocurre un error en BWAPI del tipo “dirección de memoria no válida”. Nuestra suposición inicial es que, por alguna razón, en el primer frame no está toda la información disponible, o al menos no disponible correctamente. Hemos intentado realizar esperas durante algunos frames pero parece no funcionar, haciendo pausas al inicio, empezar con

velocidades más lentas, y no conseguimos que funcionara. En concreto, hemos tenido este error en dos situaciones diferentes:

  - Cuando se intenta realizar una acción de ataque, se buscan unidades enemigas, haciendo uso de la función *isAlly* de BWAPI. Esta función provoca el error mencionado anteriormente.
  - Trabajando con grupos de unidades, por la misma razón de que parece ser que toda la información no está disponible, provoca que no se encuentren todas las unidades, y por tanto no se generen los grupos de manera adecuada.

- Conseguir que el agente de grupo de unidades funcione correctamente: en parte por los problemas de la parte anterior, nos ha sido imposible conseguir hacer experimentos con los grupos, más allá de conseguir agrupar unidades y que se muevan de manera independiente.
- Crear más acciones, por ejemplo para los grupos hacer acciones del tipo "atacar todos a la unidad enemiga más fuerte" o "cambiar la unidad propia que está recibiendo el daño para intentar que no muera".
- Conseguir un mejor sistema para realizar jerarquías entre los agentes.

Respecto al aprendizaje por refuerzo:

- Investigar nuevos métodos de aprendizaje como por ejemplo algoritmos evolutivos o análisis de partidas y hacer una comparación de los resultados.
- En las funciones de temperatura, hacer que el potencial dependa de las recompensas que se van obteniendo, por ejemplo que esté relacionado con la distancia de la recompensa que se está obteniendo últimamente con una recompensa optimista, de tal forma que cuanto menor sea esta distancia, la épsilon crezca más rápido, y alpha disminuya más rápido.
- Mejorar el rendimiento. En especial, en la Q-Tabla se guardan todos los estados, y además las trazas de elegibilidad se tratan de una manera muy ineficiente. Hay varias alternativas: agrupar estados similares para que se traten igual, crear los estados sólo cuando se visiten por primera vez (evitar tener los estados por defecto que nunca se visitan).
- Investigar nuevas representaciones de los estados, por ejemplo en lugar de tener en cuenta la distancia al enemigo, tener en cuenta también su posición. Esto se podría realizar con algo parecido a una matriz de peligro alrededor de la unidad aliada, de tal forma que si dividimos su rango de visión en, digamos, una matriz de 3x3, en cada casilla se evalúe el peligro que corre por estar en la misma, y tomar acciones en base a ello.
- En cuanto a la coordinación de grupos de unidades, o más concretamente, coordinación de agentes la idea más sencilla sería crear una jerarquía de agentes, por ejemplo un agente "ejército" que tome decisiones del tipo "grupos de unidades 1 y 2 se colocan delante y atacan a lo más cercano" y

> “grupo de unidades a larga distancia 3, se quedan atrás disparando a los enemigos más lejanos”, de tal forma que indica a los agentes 1 y 2 moverse hacia adelante, y al 3 detrás, y cuando terminen de colocarse les ordena atacar, cada uno según haya aprendido.

Como reflexión final, cabe notar que aunque es un campo bastante complejo, el aprendizaje por refuerzo es un gran método de aprendizaje y bastante potente, que se aproxima bastante a los métodos de la psicología y la biología. Desde un punto de vista computacional, nos parece una gran aproximación al problema del aprendizaje automático que cubre algunas de las deficiencias de los grandes métodos de aprendizaje supervisado y no supervisado, como el hecho de no necesitar un gran set de datos para ser entrenado, siendo no un sustituto para estos métodos, pero si un método complementario bastante potente, ya que al aprender de la experiencia, puede mejorar poco a poco en casi cualquier campo en el que se aplique a medida que se ejecuta.

Los tres coincidimos en el hecho de que sería bastante interesante estudiar aprendizaje por refuerzo en alguna asignatura, ya que es algo que hemos tenido que estudiar por nuestra cuenta, y aunque hemos cursado las asignaturas de inteligencia artificial y de aprendizaje automático, es algo que solamente se menciona como alternativa a los métodos más comunes o de lo que se ve una descripción teórica sin llegar a profundizar en ello ni a ver ejemplos de algoritmos o aplicaciones.

Respecto a la aplicación del aprendizaje por refuerzo en StarCraft, y en videojuegos en general, pensamos que es un campo en el que se puede profundizar bastante, y del que se pueden obtener resultados bastante satisfactorios, permitiendo juegos que se adapten al propio jugador, variando la dificultad en función de sus acciones, y mejorando la experiencia del mismo.

Aprovechamos para recordar que el código de nuestro framework se encuentra en GitHub con licencia copyleft, y que puede ser utilizado tanto para ser mejorado, como para futuros trabajos, adjuntamos de nuevo la página aquí:

https://github.com/TFG-StarCraft/TFG-StarCraft

Agradecemos al lector su interés y dedicación a la hora de leer nuestro documento, y le deseamos lo mejor.

Capítulo 7. Aportaciones individuales

7.1 Miguel Ascanio Gómez

En general he participado en todos los aspectos del proyecto, especialmente en la implementación y en el diseño, tanto de algunos de los experimentos, como del *framework*.

Por la parte del Q-Learning, me encargué de implementar los algoritmos en Java. Inicialmente, una implementación del Q-Learning más básico, sobre un array estático de tamaño fijo, sólo válido para un laberinto, para la primera prueba con StarCraft.

Después vino la mejora de dicha implementación, primero pasando a utilizar un HashMap y no un array estático; y luego mejorando el diseño de la estructura para que fuese más sencillo de extender y mejorar.

Posteriormente, cambiar a TD(λ) implicó realizar más cambios: por un lado, incluir la tabla de la E, y por otro, realizar los cambios pertinentes en el propio algoritmo (las modificaciones a la E, la nueva forma de calcular la Q en base a la E). Al final, en el código quedan implementadas varias alternativas: por un lado, tabla hash con la Q y la E; y por otro una con solo la Q, en caso de querer hacer pruebas con el algoritmo antiguo.

En cuanto a BWAPI, ayudé en la parte de entender cómo funciona la API, y sobre todo como explotarla adecuadamente en nuestro *framework*. También me encargué de realizar las actualizaciones de nuestro código según se actualizaba la propia BWAPI, para trabajar con la versión más actualizada de la misma.

También me encargué del diseño de todo el *Framework*, además de buena parte de la implementación. Inicialmente, lo que íbamos programando, aunque funcionaba, quedaba un poco sucio, y siempre que había que cambiar algo era un verdadero dolor de cabeza. Así que por el bien del proyecto, decidimos que lo mejor sería establecer una buena estructura. En general, el código se fue desarrollando en fases:

- Al principio, para los experimentos con laberintos el código era bastante simple y difícil de manejar, como ya comento más arriba. Sin embargo, ya presentaba una característica clave de todo el framework: la existencia de dos threads (que luego se ampliaría al añadir varios agentes), un thread para la comunicación con BWAPI, y otro thread para la lógica del Agente y del motor de toma de decisiones. Al principio hubo muchos problemas de sincronización, que fueron paulatinamente reparados, hasta la versión final, donde cambié toda la estructura de sincronización por una más sencilla, efectiva y sin fallos (conocidos).
- Posteriormente, vino uno de los mayores cambios del código: pasar de un dominio tan simple como un laberinto, a algo más complejo que tenga en cuenta

posiciones y puntos de vida relativos. Aquí opté por modificar el código para que funcionase de manera genérica a cualquier representación del estado que deseásemos, no hacerlo específico para posición relativa y vida, ya que si quisiéramos cambiarlo volvería a ser un dolor de cabeza. Esta implementación genérica me llevó a cambiar la estructura del código, en especial:

  - Por un lado, establecer una estructura adecuada a la representación de estados, para poder crear nuevas representaciones fácilmente, con un número indeterminado de dimensiones.
  - Por otro, tanto el sistema de toma de decisiones tiene que ser independiente de cómo se implemente la representación del estado. Para ello, una clase abstracta, de la que extiende cualquier representación, permite al *DecisionMaker* actuar independientemente de cómo esté implementado la representación, pues el *DecisionMaker*, al fin de al cabo, sólo necesita los valores discretos de las dimensiones de la representación del estado.
  - En cuanto al Bot/Agente y a las acciones, son *en parte* independientes de la representación del estado. Bien es cierto que si cambiamos las dimensiones no hay nada que cambiar; pero entre el problema del laberinto y el de matar a enemigos hay tres cosas que cambian: la representación del estado (arreglada en el punto anterior), las acciones a tomar, y el cálculo de recompensas y decisión del estado final. De esta forma, también es interesante garantizar que se puede cambiar tanto de acciones como de cálculo de recompensas fácilmente. El primer punto es fácil: diseñando un sistema de selección de acciones en la GUI; el segundo tampoco tiene mucha ciencia: en la implementación inicial esto se realizaba en el *Decision Maker*, en el primer cambio esto se pasó al Bot, que quedaba como clase abstracta, y las implementaciones de la misma definirían el cálculo de recompensas y la definición del estado final; y en la implementación final, esto quedaría entre los agentes y el agente master.

- Finalmente, el siguiente cambio vino propiciado por nuestra necesidad de extender el sistema para controlar grupos de unidades. Todo este sistema es el definido en el [Capítulo 4] de este documento. Respecto a la implementación anterior, hubo bastantes cambios, aunque más de la estructura que de la lógica:
  - La parte más modificada fue la parte de bajo nivel (Bot). Anteriormente sólo existían el Bot, las Acciones (tanto del bot como del *Decision Maker*), y el *Decision Maker*). Esto lo extendí incluyendo los agentes (que cumplían básicamente la función de las clases que extendían al bot en la implementación anterior, calculando la recompensa y decidiendo el estado final).

- También hubo que cambiar algo del *DecisionMaker*: por un lado, antes sólo existía una instancia, ahora existe un decision maker por cada agente en el sistema; por otro, en caso de haber varios agentes trabajando sobre la misma tabla, hay que arbitrar cuál de ellos escribe.
- Y finalmente, hubo que cambiar la parte de la sincronización, para garantizar que todos los agentes se comuniquen adecuadamente con sus *Decision Makers*.

En general, creo que el código es bastante útil, en lo que se refiere a utilizarlo para futuros proyectos. Sin embargo, no está acabada, y no está falta de errores, de hecho, no hemos conseguido que funcione bien con grupos de unidades, como se detalla en el apartado Trabajo Futuro. Aun así, creo sinceramente que para hacer experimentos con BWAPI, es más sencillo trabajar desde nuestro *framework* arreglando sus errores y extendiéndolo lo que sea necesario, que empezar de 0 sobre BWAPI.

En cuanto a la parte de diseño de experimentos y pruebas, ayudé a diseñar algunos, como por ejemplo algún laberinto, o algunas pruebas para los grupos de unidades. En general, me centré en buscar algunos experimentos adecuados a nuestro trabajo, de tal forma que lo que teníamos implementado se adecuara al experimento diseñado.

En cuanto al desarrollo de este documento, me encargué del capítulo dedicado a la descripción del *framework*, así como, junto a mis compañeros, la parte de las conclusiones y trabajo futuro.

7.2 Alberto Casas Ortiz

En cuanto al estudio del algoritmo Q-Learning y del aprendizaje por refuerzo, una de mis principales aportaciones al proyecto fue el desarrollo de las diferentes aplicaciones laterales para ayudar a entender mejor la naturaleza del algoritmo Q-Learning.

En primer lugar, cuando empezamos a utilizar el algoritmo Q-Learning, desarrollamos entre los tres una pequeña aplicación que resolvía un laberinto sencillo hecho en java. Viendo que podría ser bastante útil a la hora de probar nuevas características o configuraciones de manera rápida y sencilla, desarrollé una nueva versión de esta aplicación y avancé más allá (Capítulo 3 - Primer experimento), añadiendo características como la función de temperatura y desarrollando una interfaz gráfica para el mismo, que más adelante serviría como modelo para la interfaz del Framework de Q-Learning que hemos desarrollado (Capítulo 4 - Framework Q-Learning). A partir de esta aplicación de laberinto descubrimos bastantes aspectos de la naturaleza del Q-Learning, entre ellos algunos bastante importantes como diferentes nociones sobre la explotación de los caminos aprendidos y exploración de nuevos caminos, y como una buena coordinación entre ambos nos da mejores resultados. También nos sirvió para buscar formas de mostrar la tabla Q(s, a) del algoritmo de una forma amigable y entendible por un ser humano.

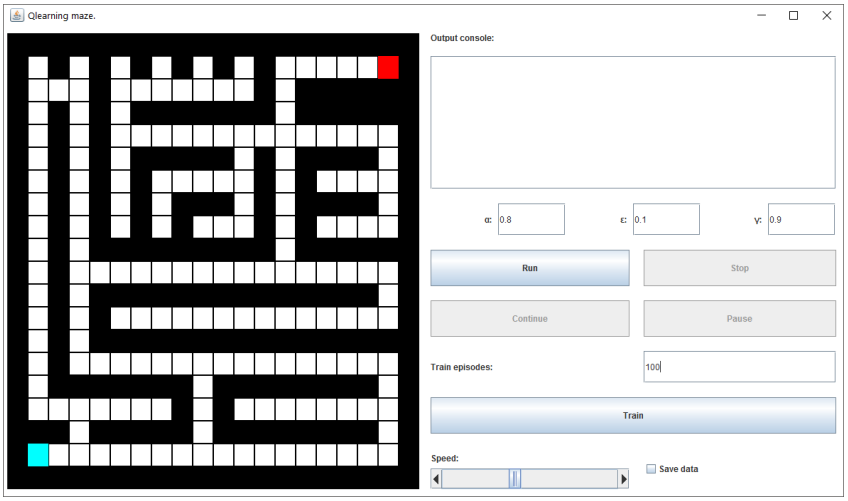


Figura 55 – GUI del laberinto

Más adelante, cuando hicimos el cambio del algoritmo Q-Learning al algoritmo Q-Learning(λ), decidimos investigar sobre los diferentes tipos de adversarios que podríamos tener, así como la manera en la que estos afectan al aprendizaje. Esto me llevó a la idea de probar adversarios en juegos sencillos, viniéndome a la cabeza el clásico 3 en raya (tic-tac-toe), por lo que desarrollé una pequeña aplicación para este propósito que aprendiera a jugar a 3 en raya contra diferentes tipos de adversarios. Esto nos dio las conclusiones que aparecen en el apartado adversarios del capítulo 2, así como la noción de que las recompensas negativas pueden afectar al propagarse a estados anteriores en los que se tomó una buena acción, empeorando su valor en la tabla Q(s, a), haciendo que el aprendizaje no sea correcto.

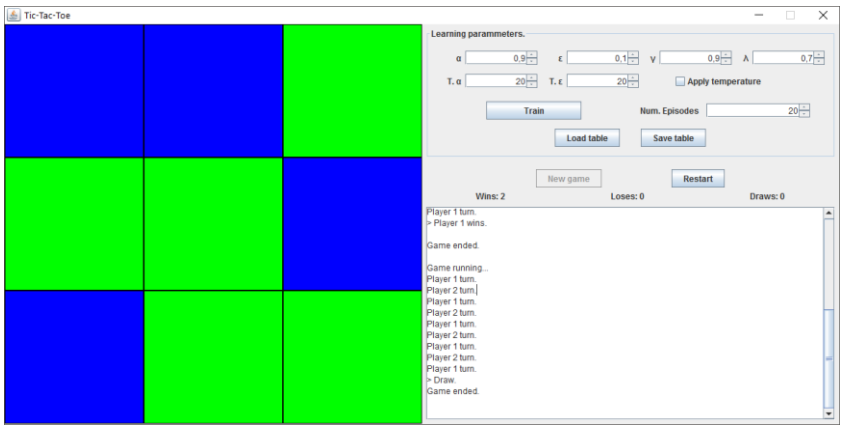


Figura 56 – GUI de 3 en raya

Respecto al framework que hemos desarrollado, mis principales aportaciones han sido en la GUI, ya que era el que más experiencia tenía en el desarrollo de éstas, y estaba basada en una de mis aplicaciones. Una de mis aportaciones fue el sistema de pestañas que utilizamos para cambiar de vistas y mostrar más información, así como algunas de esas pestañas, como units, que mostraba la posición y rango de las unidades con

respecto a nuestro agente, o la idea de incluir gráficas directamente en la GUI con jmathplot. También desarrollé una serie de funciones relacionadas con la detección de otras unidades alrededor del agente y determinar características como si son aliadas, enemigas, el centro de un grupo o incluso estimar si una unidad es más fuerte que nosotros y puede vencernos. Otra de mis aportaciones fue el desarrollo de acciones del agente relativas a otras unidades, como rodear una unidad o grupos de unidades, acercarte o alejarte de aliados o enemigos.

En lo que respecta a la memoria, todos hemos participado en ella, ya que nos dividimos el trabajo. Una de mis tareas a la hora de redactar la memoria fue la traducción de algunos apartados a inglés. También redacté el capítulo 3 ya que estaba basado en una de mis aplicaciones y algunos apartados del capítulo 2, como las explicaciones de que es el aprendizaje por refuerzo con respecto a los otros dos grandes tipos de aprendizaje o la explicación del algoritmo Q-learning y el problema del equilibrio entre exploración y explotación, y del capítulo 5, como la explicación de la destrucción de objetos móviles y del sistema curar-atacar, realizando múltiples ejecuciones para sacar conclusiones lo más correctas posibles y poder plasmarlas en gráficas. También, como me encargué de la interfaz gráfica, redacté la parte pertinente a este componente en el capítulo 4.

Otro de mis cometidos fue dar un formato entendible a la bibliografía, ordenada por orden alfabético y con referencias numéricas. Intenté que la bibliografía siguiera un formato standard, cosa difícil al incluir no solo libros y artículos, sino también links a webs. La parte de conclusiones y trabajo futuro la hicimos juntos entre los tres, ya que es un apartado al que todos teníamos algo que aportar.

Mi último cometido consistió en dar formato a toda la estructura de la memoria, normalizando lo escrito por todos y estableciendo el formato para las figuras e imágenes que aparecen.

7.3 Raúl Martín Guadaño

Mis aportaciones a este trabajo se han centrado en investigación y búsqueda de errores, así como de la obtención de datos.

Cuando empezamos a trabajar el Q-Learning, estuve investigando acerca de este algoritmo para entender su funcionamiento, así como leyendo el trabajo realizado por otros alumnos el año pasado acerca de aprendizaje por refuerzo en Starcraft. Además de esto, busqué y leí artículos de aprendizaje por refuerzo en videojuegos para coger ideas que pudiésemos aplicar en nuestro proyecto a la hora de presentarlo o para trabajo futuro una vez entregado el proyecto. Con todo esto, obtuve el conocimiento necesario para poder trabajar cómodamente en el proyecto, sin tener que parar a revisar el algoritmo o sus parámetros para poder utilizarlos en cada problema que proponíamos.

Nuestra forma de trabajo en grupo para este proyecto comenzó con una reunión semanal larga en la que intentábamos desarrollar un nuevo problema. En estas reuniones solo se trabajaba en un ordenador de forma que ninguno de los integrantes se quedase atrás en el proyecto y todos entendiésemos y conociésemos el código. En estas reuniones (que siguieron llevándose a cabo hasta final de curso), el más veloz de nosotros era quien programaba mientras que el resto intentábamos aportar ideas y ver posibles errores que mientras que se programa no se ven tan claros. Por este motivo, una gran parte del trabajo realizado en el proyecto se realizó en grupo y no de forma individual.

Debido a mi forma lenta de programar, mi dedicación principal en el proyecto ha sido esta misma, localizar errores y tratar de proponer ideas para solucionar los mismos. Para realizar la búsqueda de errores, realizaba pruebas en casa en las que mediante depuración intentaba localizar los fallos del código que hacían que nuestro programa no funcionase correctamente. También me encargaba de observar la evolución del aprendizaje de nuestra unidad de combate a lo largo de los episodios para corroborar que aprendía a medida que jugaba partidas y que los valores que se tomaban de las constantes del algoritmo eran los correctos.

Previamente a la búsqueda y eliminación de errores, a la hora de instalar la BWAPI, en mi ordenador se producía una violación de segmento al intentar ejecutar el videojuego sincronizado con la BWAPI, para lo cual tuve que finalmente formatear el ordenador, asi que no pude aportar mucho fuera de las reuniones semanales hasta que pude instalarlo correctamente. En ese tiempo estuve realizando la tarea de investigación de artículos publicados nombrada anteriormente, así como un estudio de la BWAPI para entender su funcionamiento y prevenir posibles fallos que pudiesen darse con respecto a ella.

Para la memoria, intenté obtener unos buenos resultados que plasmar en la misma. Empecé por redactar la introducción al proyecto en español incluyendo todos sus apartados de forma resumida; introducción, motivación, objetivos y estructura del documento, para más tarde expandirlos y completarlos. Después, realicé el apartado del algoritmo Q-Learning (λ), para el cual tuve que realizar un resumen de toda la información extraída de los libros y artículos utilizados para el proyecto, así como la explicación de su pseudocódigo. Finalmente, realicé numerosas pruebas para obtener datos que exponer en el apartado de problemas propuestos. Por mi parte realice los dos primeros problemas: resolución de laberintos y destrucción de objetivos móviles.

- Resolución de laberintos. Para este problema, tuve que volver a la versión 3.7.5 de la BWAPI, ya que fue la versión que utilizamos en este primer problema. Para este problema realicé pruebas en los 3 mapas que desarrollamos y guardé los datos obtenidos para trabajar con ellos. Con esos datos, realicé gráficas movimientos/iteración para tener una perspectiva mas visual de esos datos y

poder comentar los picos que se generaban. Finalmente, recalqué los datos más significativos obtenidos y realicé una conclusión para ese problema.

- Destrucción de objetivos inmóviles. En este caso, la versión de BWAPI utilizada era la 4.0.1.b y tuve que instalarla para realizar la prueba tal y como se realizó en su momento. Para este problema utilizamos dos mapas, uno con una estructura que no atacaba y otro con una que sí lo hacía. En este caso, los datos obtenidos no eran tan interesantes como en el problema anterior. El número de movimientos ya no era importante y simplemente realicé una explicación del problema y su resolución, con las conclusiones obtenidas y la evolución hasta el siguiente problema propuesto.

Finalmente, realicé junto con mis compañeros las conclusiones del proyecto, así como los posibles trabajos futuros para el proyecto.

Bibliografía

1. **Andrew Ng, Coursera** - Aprendizaje Automático. https://es.coursera.org/learn/machine-learning
2. **Borrajo, D., González, J., Isasi, P**. - Aprendizaje automático. Sanz y Torres, 2006
3. **BWAPI, Brood War API** - https://github.com/bwapi/bwapi
4. **BWMirror, Brood War Mirror** - https://github.com/vjurenka/BWMirror
5. **JavaNativeAccess** - https://github.com/java-native-access/jna
6. **JMathPlot** - https://github.com/yannrichet/jmathplot
7. **Luger, G.F.** - Artificial Intelligence. Addison-Wesley, 2005, 5ª edición.
8. **Palma, J. T., Marín, R. -** Inteligencia Artificial. McGrawHil, 2008
9. **Rich, E. y Knight, K.** - Artificial Intelligence. McGraw-Hill, 1991, 2ª edición.
10. **Richard S. Sutton, Andrew G. Barto** - Reinforcement Learning: An Introduction. The MIT Press, 2012.
11. **Russell, S. y Norvig, P** - Inteligencia Artificial: Un Enfoque Moderno. Prentice Hall, 2004, 2ª edición.
12. **Sierra, B**. - Aprendizaje automático. Pearson Prentice-Hall, 2006
13. **Skinner, B.** - Sobre el conductismo. Planeta-De Agostini, 1974.
14. **Starcraft: Brood War** - http://us.blizzard.com/en-us/games/sc/
15. **Udacity** - Machine Learning: Reinforcement Learning - https://www.udacity.com/course/machine-learning-reinforcement-learning--ud820
16. **Watkins, C.J.C.H.** - Learning from Delayed Rewards. PhD thesis, Cambridge University, 1989.
17. **Weka** - http://www.cs.waikato.ac.nz/ml/weka/
18. **Witten, I.H., Frank, E., Hall, M.A.** Data Mining. Morgan Kaufmann, 2011, 3rd edition
19. **Juan A. Botía Blaya** Aprendizaje por Refuerzo Tratamiento Inteligente de la Información y Aplicaciones